UNIVERSITÉ DE NAMUR

Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche

researchportal.unamur.be

University of Namur

THESIS / THÈSE

MASTER IN COMPUTER SCIENCE

Paraphraser for Albert II requirements specifications

Beirekdar, Abdo

Award date:
1998

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 26. Apr. 2024

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'Informatique

Année académique 1997-1998

Paraphraser for AlbertII requirements specifications

Abdo BEIREKDAR

Mémoire présenté en vue de l'obtention du grade de Licencié en Informatique.

Acknowledgments

I am very happy to be able to thank here all those who helped me to do this work:

My promoter Eric Dubois, who followed me in my work and helped me by reading and commenting this document many times during its elaboration.

Patric Heymans whose conversation permitted a better comprehension of the AlbertII language, and who supported me in the realization of this work.

I would also like to thank Michaël Petit who very kindly helped me to realize the case study of this work.

Finally, I would like to thank the atomic energy commission of Syria, without whom my study in Belgium would not have been possible.

Abdo.

Table of Contents

INTRODUCTION 1

CHAPTER ONE : NATURAL LANGUAGE GENERATION (NLG) 5

1.2 MAIN NLG TECHNIQUES 5
1.1 KNOWLEDGE-BASED SYSTEM : KBS 5
1.2 TEMPLATE-BASED SYSTEMS : TBS 7
1.3 COMPARISON BETWEEN KBS AND TBS 8
1.3.1 Advantages of KBS *8*
1.3.2 Advantages of TBS *8*
1.4 HYBRID SYSTEMS 9
1.5 APPROACH USED IN THE ALBERT II PARAPHRASER 9

CHAPTER TWO : ALBERTII LANGUAGE 11

2.1 THE RUNNING EXAMPLE 12
2.2 PARAPHRASED OBJECTS OF AN ALBERT II SPECIFICATION 12
2.2.1 Basic Data Types *12*
2.2.2 Constructed Data Types *12*
2.2.3 Operations *13*
2.2.4 Society Definition *13*
2.2.5 Agent States Components *14*
2.2.6 Agent actions *14*
2.3 NON TREATED SECTIONS 15
2.3.1 Derived components *15*
2.3.2 Initial valuation *15*
2.3.3 State behavior *15*
2.3.4 Actions composition *15*
2.3.5 Actions duration *16*
2.3.6 Actions precondition *16*
2.3.7 Effects of Action *16*
2.3.8 Triggerings *16*
2.3.9 Actions perception *17*
2.3.10 State perception *17*
2.3.11 Actions information *17*
2.3.12 State information *17*
2.4 LINGUISTIC INFORMATION 18
2.4.1 Necessity of linguistic information *18*
2.4.2 Formal objects related to ENTITIES *18*
2.4.3 Formal objects related to ACTIVITIES *19*
2.4.4 Linguistic precessions *20*
4.4.5 Providing linguistic information *20*

CHAPTER THREE : THE PARAPHRASING PATTERNS 21

3.1 INDIVIDUAL PARAPHRASING PATTERNS 21
3.1.1 Conventions *21*
3.1.2 Basic data type *21*
3.1.3 Redefined data type *21*
3.1.4 Cartesian Product data type *21*
3.1.5 Set/seq/bag data type *22*
3.1.6 Table data type *22*
3.1.7 Union data type *22*
3.1.8 Enumerated data type *22*
3.1.9 Operations *23*
3.1.10 Society composition *24*
3.1.11 Set/Sequence State component *24*
3.1.12 Table State component *24*

3.1.13 Instance State component *24*
3.1.14 Derived component *25*
3.1.15 Exported component *25*
3.1.16 Constant component *25*
3.1.17 Actions *26*
3.1.18 Exported action *26*
3.1.19 Combined actions *26*
3.2 GLOBAL PARAPHRASING OUTPUT 26

CHAPTER FOUR : PARSING BASICS **27**

4.1 LEXICAL ANALYSIS 27
4.1.1 Regular expressions *28*
4.2 SYNTAX ANALYSIS 28
4.2.1 Context-free grammars *29*
4.3 BASIC PARSING TECHNIQUES 30
4.4 BUILDING A PARSER WITH VISUAL PARSE++ 31
4.4.1 The Lexical analyzer in Visual Parse++ *31*
4.4.2 Syntax analyzer in Visual Parse++ *33*

CHAPTER FIVE : THE ALBERTII PARAPHRASER **37**

5.1 THE PARAPHRASER REQUIREMENTS 37
5.2 THE PARAPHRASING PROCESS 37
5.3 THE LOGICAL ARCHITECTURE 38
5.3.1 Function COORDINATOR *39*
5.3.2 Type PARSETREE *46*
5.3.3 Type DICTIONARY *48*
5.3.4 Type PARABASE *48*
5.4 THE PHYSICAL ARCHITECTURE 49
5.4.1 Unit COORDINATOR *49*
5.4.3 The dictionary/ The parabase types *54*
5.5 THE PARSER IMPLEMENTATION 54
5.5.1 Tokens of the lexical analyzer *54*
5.5.2 Description of the grammar *55*
5.5.3 The parser *58*
5.5.4 Realization *60*

CONCLUSION **61**

REFERENCES **63**

ANNEXES **65**

Abstract

The objective of this work is to develop and implement an interactive program to paraphrase a part of a specification written using the Albert II language. The result is a text in **English** generated under two forms according to the person concerned with this text. If he is familiar with the AlbertII language, the text can be just the paraphrase of the formal semantic of the specification, without adding any information to it. In this case, the objective of such a text is to facilitate the revision of the specification by taking off all the comments automatically provided by the editor of the language (e.g. headings of different sections of the specification). If the person concerned with the generated text is a non-expert one, the program gives the analyst the possibility to provide more information to explicit the real semantics of his specification (significant names and/or definitions for objects). In the two cases, the result of the paraphrasing process can be generated in an independent file, or with the specification by inserting the paraphrase of an object as a comment of this object. This resulting file is generated under two formats : a Text ASCII format, or a Hypertext format.

The part of specification handled by the program includes the following Albert objects: basic types, constructed types, operations on types, societies and agents declarations, state components and actions of agents.

Resumé

L'objectif de ce travail est de réaliser un programme interactif qui permet d'analyser une spécification écrite en langage Albert II et de paraphraser cette spécification en un texte en **Anglais**. Le texte résultant peut être généré sous deux formes selon la personne à laquelle le texte est destiné. S'il est destiné au concepteur de la spécification, ou à une personne familier avec le langage AlbertII, le texte généré peut être simplement la paraphrase liée à la sémantique formelle de la spécification, sans y ajouter aucune information. L'objectif de générer un tel texte est alors de faciliter la révision de la spécification en enlevant toutes les commentaires produits automatiquement par l'éditeur du langage Albert (ex. entêtes des différentes sections de la spécification). Si le texte généré est destiné à une personne non-expert, le programme donne au concepteur de la spécification la possibilité de donner des informations complémentaires (des noms signifiants au objets de la spécification et des définitions de ces objets) pour expliciter la sémantique réelle de sa spécification. Dans les deux cas, le résultat de paraphrase peut être généré dans un fichier indépendant, ou avec la spécification en insérant la paraphrase d'un objet comme un commentaire de cet objet. Ce fichier résultant est généré sous deux formats : un format Texte ASCII, ou un format Hypertext.

La partie de spécification traitée comprend les objets Albert II suivants : les types de base, les types construits, les opérations sur les types, la déclaration des sociétés et des agents, les éléments d'états et les actions des agents.

Introduction

Introduction

The term Requirements Engineering (RE) refers to this part of the system development cycle investigating the problems and requirements of the users community and developing a specification document of the future system.

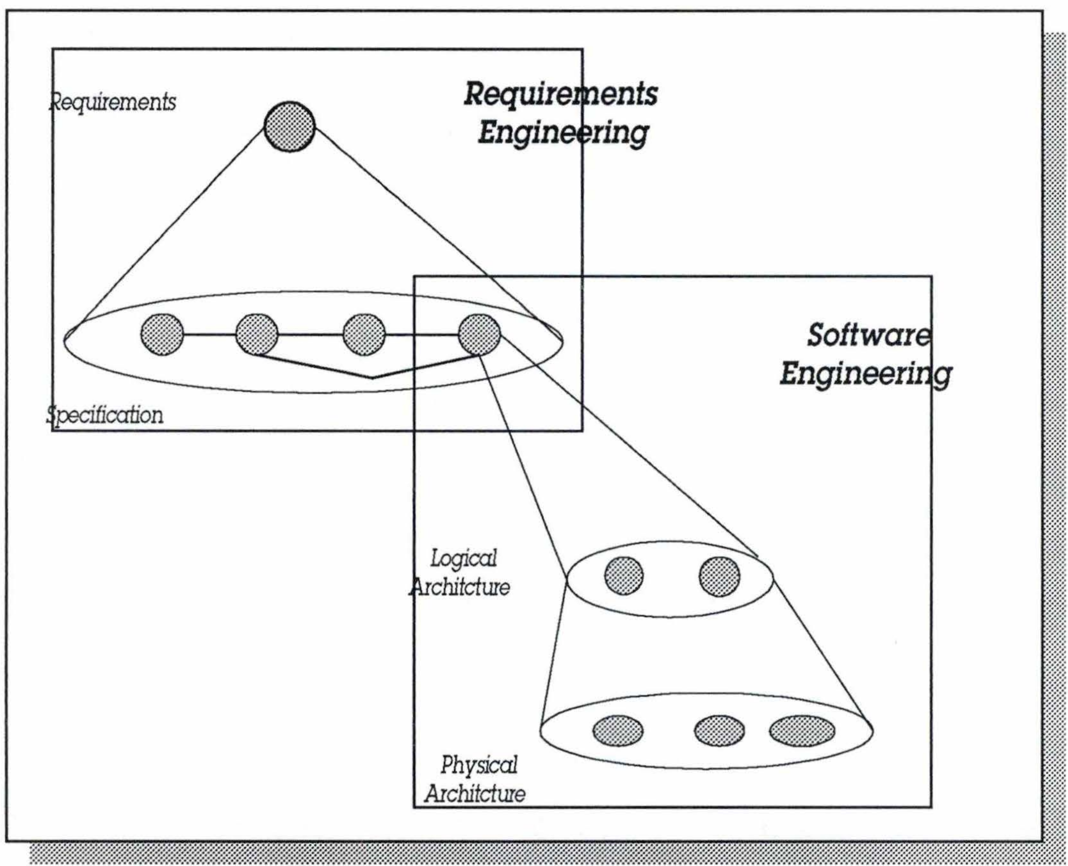

Figure 0.1: system development cycle

RE is an important phase because the resulting document plays a central role since it is part of the contractual agreement taking place among customers, analysts, designers, etc. The errors in the formulation of requirements are considered as the most costing [Dubois97], and can have bad consequences on the project progress, on its objectives and on customers satisfaction.

Error type	%Total errors	Cost relative correction	%Cost total correction
Specification	40%	5	66%
Conception	30%	2.5	25%
Coding	30%	1	9%

Figure 0.2: Errors cost

Taking into consideration the variety of stakeholders and the diversity of their culture, the requirements document has to be written using a common notation readable by all of them.

Analysts can be supported in their activities by the use of conceptual modeling languages (semi-formal approach) or by requirements specification languages (formal approach). But notations supported by these languages are poorly readable by the other stakeholders, in particular the customers.

Hence the necessity of paraphrasing the requirements document into a natural language text making it the base of communication among all stakeholders.

Our work can be defined as an attempt to implement a paraphraser that generates a text in English from a piece of formal specification written in the Albert II[1] specification language [Du97]. The paraphraser generates two forms of text : the **analyst-oriented** form contains a simple paraphrase of the specification semantic as it is formally defined by the Albert II language. The **customer-oriented** form contains information provided by the analyst to explicit the real semantic of his specification : he can give significant names of objects (e.g. 'Address' instead of 'Adr'), objects definitions (e.g. 'Adr' denotes "customer work address") or even give his proper comment to replace the comment generated by the program concerning some objects.

The input of the paraphraser is a specification file produced by the language editor, or produced by any other text editor and respecting the Albert II editor conventions[2]. The output is an English text. When this output is Customer-oriented, the input might include in addition to the specification file, some more information provided by the analyst to enable the generation of an output useful for the customer.

The paraphraser is implemented under Microsoft Windows95[3] environment, using the Borland Delphi[4] programming language. We used Visual Parse++[5] to write the grammar of the parser used to analyze the input specification.

The rest of this presentation will be as follow :

- In **chapter One,** we are going to give a brief overview of the main techniques of natural language generation.
- In **chapter Two**, will be devoted to the presentation of the Albert II language through the handling of a very simple example. We will indicate the formal objects of the language our work is restricted on, and suggest the linguistic information used for paraphrasing these objects.
- In **Chapter Three**, we will give in details the suggested paraphrases for the treated formal kinds of Albert II objects, and the global paraphrase of the treated specification.
- Recognizing and extracting the Albert II objects in the treated specification, is the first major task of our paraphraser, so in **Chapter Four** we will present a theoretical overview of major parsing concepts: parsers, lexical analysis and syntactical analysis. We conclude this chapter by presenting the Visual Parse++ that we used to build our parser.
- In **Chapter Five**, we present in details the method followed to realize our paraphraser.
- The **Conclusion** will be devoted to comments on the implementation of the paraphraser, and to conclude by giving our opinion about the exposed paraphrasing attempt and by indicating some directions of work for the future.

---

[1] ALBERT (Agent-oriented Language for Building and Eliciting Real-Time requirements).

[2] The Albert editor automatically generates headings for the different specification sections and some special characters before some key words that are not mentioned in the syntax of the language.

[3] A product of Microsoft Corporation.

[4] A product of Borland Corporation.

[5] A product of SandStone technology Incorporation.

- In annex, we joint the specification and the paraphrase of a very simple case study, and the main units of the developed paraphraser.

Chapter One

Natural Language Generation (NLG)

Chapter One

Natural Language Generation (NLG)

In natural language generation, a computer automatically creates natural language, e.g. English, French, or Arabic, from a computational representation. One use of Natural Language Generation is to describe software systems from a formal specification of the software system. Most people do not understand formal languages, but they understand natural languages, therefore it is desirable to have a tool which automatically generates natural language from a formal specification.

Other fields for the application of natural language generation are, e.g. automatic technical documentation generation, automatic weather reports from raw data, explanations in expert systems, medical informatics and machine translation between natural languages and translation to multiple natural languages from a source representation.

In [Robin98] we find a good overview of the NLG history from the early 70s where the first domain-based NLG applications appeare, to the early 90s where we find the first industrial applications.

1.2 Main NLG techniques

The main two techniques used in NLG applications are the knowledge-based (KB) technique that have been studied by the NLG community, and the so called template-based (TB) technique, that simply manipulate character strings, in a way that uses little, if any, linguistic knowledge [Reiter95].

One important question when applying NLG is what benefits each of these two approaches offers with respect to the other, and when it is the most useful. We will not try to answer this question, but we will give a brief presentation of these two kinds of NLG techniques, then, we will depend on this presentation to justify the choice that we have made to realize our paraphraser.

1.1 Knowledge-Based System[1] : KBS

A knowledge base can be defined as the set of skills and judgments that we have and apply to the information in our domain. For example, a financial knowledge base could be formed of the rules and regulations set out by the government's tax and other legislation and other policies.

In a KBS, the knowledge is made explicit, rather than being implicitly mixed in with the algorithm according to which the data is treated. When this is done, the algorithm also has to include a reasoning or inference mechanism so that deductions and conclusions can be drawn from the knowledge. Thus, a KBS can be seen as:

PROGRAM (KBS) = EXPLICIT KNOWLEDGE + REASONING MECHANISM+ DATA.

---

[1] Also called expert systems.

The most common and simplest way of representing knowledge is in the form of rules - this type of programming is sometimes referred to as rule-based programming. In such representation, we could have:

IF it is evening AND the sky is red THEN expect good weather tomorrow.
IF Machine_ T > 40 AND operate_T < 15 THEN stop Machine AND alert Worker.

Another way to represent knowledge is using frames. A frame is very similar to the concept of an object in object-oriented programming, but is rather more powerful and pre-dates it by about ten years[].

KBS and knowledge technology can be used for an extremely wide variety of tasks and it is more advanced programming paradigm compared with the conventional paradigm in which a system is seen as : PROGRAM = ALGORITHM + DATA, so the knowledge is not implicitly represented.

In cases where the knowledge has never before been written down but is in the head of an expert, special analysis techniques and skills are required. These are commonly referred to as knowledge elicitation techniques.

From a technical perspective, KBS perform the following three tasks :

- ***Content Determination and Text Planning***: decide what information should be communicated to the user (content determination), and how this information should be rhetorically structured (text planning). These tasks are usually done simultaneously.

For example, as an answer on the question ' Should I take AI course', ADVISOR II[1] generates the text :

AI deals with many interesting topics, such as NLP, Vision and KR.
But it has many assignments which consist of writing papers.
You have little experience writing papers.
So it could be difficult.
I would not recommend it.

- ***Sentence Planning***: decide how the text will be splited among individual sentences and paragraphs, and what cohesion devices (e.g. pronouns, discourse markers) should be added to make the text more fluent This task includes :

A) Conjunction and other aggregation. For example:
Abdo uses glasses.
Ali uses glasses.
become
Abdo ***and*** Ali use glasses.

B) Pronominalization and other references. For example:
I just saw Ali.
Ali uses glasses.
become
I just saw Ali. ***He*** uses glasses.

[1] Adviser II is a question-answering system which provides university students with advice about which courses to take in a semester [Elhadad 93].

C) Introducing discourse markers. For example:
If I go to the institute, I should see Ali.
becomes
If I go to the institute, I should ***also*** see Ali.

The common theme behind these operations is that they do not change the information content of the text, but they do make it more fluent and easily readable.

- ***Realization*** : generate the individual sentences in a grammatically correct manner. A releaser generates individual sentences (typically from a 'deep syntactic' representation). It needs to make sure that the rules of the generated text language (e.g., English) are obeyed, including :

A) Point absorption and other punctuation rules. For example, the sentence *I saw Anca, Lolo* should end in '.' not ',.'

B) Morphology. For example, the plural of *box* is *boxes*, not *boxs.*

C) Agreement. For example, *I am here* instead of *I is here.*

D) Reflexives. For example, *Abdo saw himself* instead of *Abdo saw Abdo.*

Example: Adviser-II [Elhadad93]

Communicative goal = take (Reader, AI):
AI covers many interesting topics such as NLP, Vision and Expert System. And it involves a good amount of programming, so it should be interesting.

Communicative goal = not (take (Reader, AI)):
AI covers logic, a very theoretical topic, and it requires many assignments, so it could be difficult.

1.2 Template-based systems : TBS

All NLG systems are, computer programs that run on some input data and produce an output (the text) from this data. Non linguistic ('template') text-generation is done via manipulating character strings ; the user writes a program which includes statements such as 'Use the new name of the type if it was provided by the analyst, and formal name otherwise'. This program can be written directly in a programming language such as C or Pascal. The key difference between this approach and the Knowledge-based systems is that all manipulation is done at the character string level ; there is no attempt to represent the text in any deeper way, at either the syntactic or 'text-planning' level.

According to [Reiter95], most programming languages and mail-merge environments provide very little support for manipulating texts in even the simplest 'linguistic' manner.

Mail-merge systems can have slightly more sophisticated capabilities, such as automatically capitalizing an inserted word if it is the first word of a sentence. However, even something as simple as changing pronouns according to gender needs to be explicitly programmed. Some mail-merge systems are integrated with grammar checkers that might in theory be able to handle some low-level syntactic problems such as verb agreement, *a* vs. *an*, and elimination of multiple commas ; however, current grammar checkers may not be robust enough to be able to do this in a reliable fashion.

Example: the Apple Macintosh Balloon Help system [Reiter95]. It can produce texts such as :

This is the kind of item displayed at left.
This shows that test data is a(n) Microsoft Word document.

and

This is a folder - a place to store related files. Folders can contain files and other folders.
The icon is dimmed because the folder is open.

In the first text, *test data* and *Microsoft Word* were inserted into template slots for 'file name' and 'application program'. Note the use of *a(n)* ; even this simple type of agreement is not done in the Balloon Help system. In the second text, the last sentence *(The icon is dimmed because the folder is opened)* only appears when the mouse is positioned over an opened folder ; just the first two sentences will appear if the mouse is positioned over a closed folder. This is an example of conditional text.

1.3 Comparison between KBS and TBS

1.3.1 Advantages of KBS

The most important advantages of KBS over TBS are :

A) *Maintainability* : template-based generators can be difficult to modify according to changing user needs. Making even a slight-change to the output of a template-based generator may require a large amount of recoding (of programs) and rewriting (of templates) ; in contrast, such a change may be straightforward to make in linguistically-based system.

B) *Improved Text Quality* : KBS can produce higher-quality output. This improvement arises from the three different processing stages used in most NLG systems based on KB techniques mentioned above.

C) *Multilingual output* : multilingual output can be achieved with templates. The quality of texts generated by this approach is not high, but this may be acceptable in some circumstances. At the other extreme, multilingual output could also be achieved by building several separate systems, one for each target language. Such a system would be expensive to construct and might prove difficult to maintain.

D) *Guaranteed conformance to document standards* : in many domains it is essential that documents conform to standards and rules such as 'sentences should not be longer that 20 words' or 'sentences should not contain more than three sequential nouns'. An KBS can paraphrase or reword texts to meet such constraints.

1.3.2 Advantages of TBS

TBS have some advantages over KBS:

A) KBS can not generate text unless they have a representation of the information that the text is supposed to communicate.

B) KBS also suffer from generic problems that are common to all new technologies. There are very few people who can build KBS, compared to the millions of programmers who can build TBS; there is also very little awareness of what KBS can (and cannot) do among most developers of systems. Additionally, there is very little in the way of reusable KB resources

(software, grammars, lexicons, etc.) which means that most KBS developers still have to more or less start from scratch.

C) In some cases like interactive systems, it is necessary to respect certain response-time constrains. Such constraints are generally better satisfied by TBS.

1.4 Hybrid systems

It is normal to build systems using the KB approach and TB approach is the same system. The basic goal of such systems is to use KB approach where it really adds value, and to use TB approach where KB one is not needed or would be too expensive.

1.5 Approach used in the Albert II paraphraser

We can say that our paraphraser is a hybrid system. we mainly adopted the template-based approach. We define a paraphrasing pattern for each kind of objects we are paraphrasing. Depending on the available information and on the analyst choices, these patterns are used to generate analyst-oriented or customer-oriented paraphrase.

During the generation of our paraphrases, we realize some KB tasks. In fact, once we have an Albert II specification, we already have a lot of information to realize these tasks. In particular, we are capable of performing some *sentence planning*: the aspects of aggregation and Pronominalization, and some *realization*: applying some punctuation rules, morphology and agreement.

Chapter Two

Albert II language

Chapter Two

AlbertII language

Albert II language is a formal specification language based on the concept of *agent* in terms of which one may express real-time requirements of a distributed (cooperative) system.

From an Albert II point of view, the specified system is seen as a *society* of *agents* and in some cases of sub-societies. An agent is characterized by an internal state modeled in terms of *state components*. An agent can perform *actions* describing its activities. These actions are characterized through the changes that they bring to the agent state. An agent is considered through its relationships with other agents of the society. He asks them to perform some actions, or he performs some actions on their demand, and by the information exchanged with these agents.

An Albert II specification is composed of many sections. Here, we are going to present some of these sections using an example: a small part of an engine assembly cell which is a sub-system of a truck manufacturing company. Our paraphraser doesn't handle all the sections of an Albert II specification, so, we will be interested in the sections that it covers (figure 2.1).

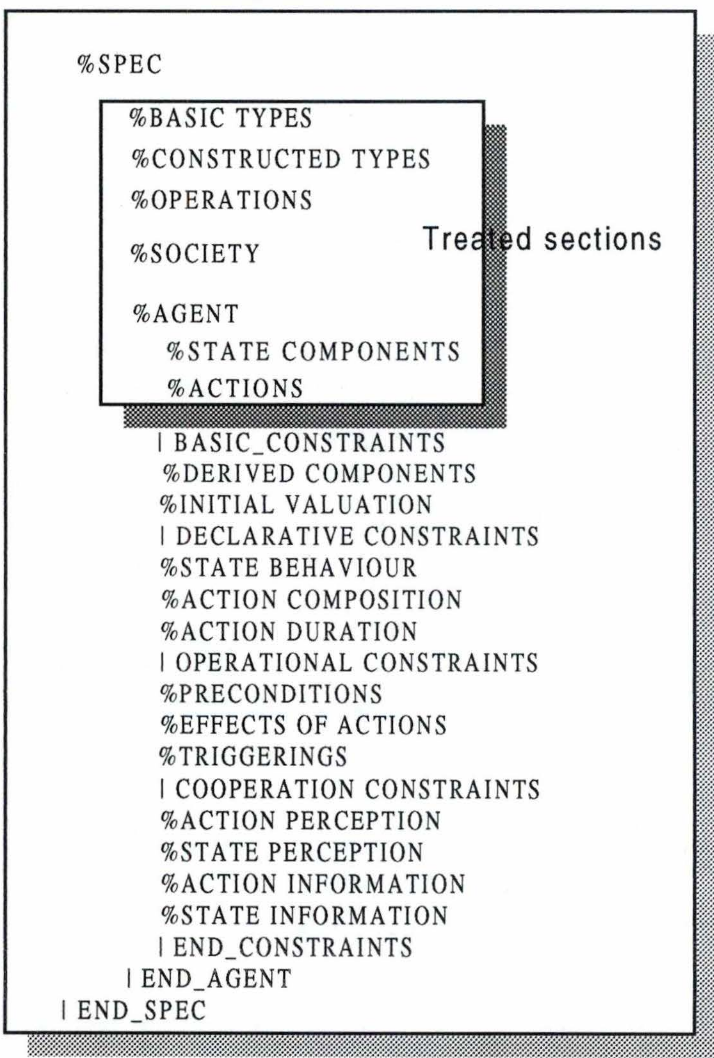

Figure 2.1 : sections and organization of an Albert II specification

2.1 The running example

We examine an engine assembly cell which is a sub-system of a truck manufacturing company. Its role is to produce engines by adding a number of small parts to a bare engine block. To some of the engines, only a part of the process is applied and the resulting engines are sent to another factory for further processing.

The cell is divided into a number of work centers. Each work center receives its necessary inputs from a transportation system. The inputs either come from a stock or from another work center.

The quantity of each kind of engine to be produced is determined by the foreman. On the basis of a production plan for the global company, he derives the daily cell production ratio. This information is then used by the cell controller to monitor the production.

2.2 Paraphrased objects of an Albert II specification

First, we are going to details the Albert II objects treated by our paraphraser. For every type of objects, we will give the formal syntax, an example of this object from our running example, and the semantic of this kind of objects.

2.2.1 Basic Data Types

In this section, we find the declaration of user defined elementary types. For example :

BLOCK_ID
PISTON
CAM_FOLLOWER

In addition to the basic types defined by the analyst, there are some types predefined in the language, these are :

- *BOOLEAN* (Boolean values, i.e. *True* and *False*),
- *CHAR* (the characters),
- *STRING* (the strings of characters),
- *INTEGER* (the positive and negative integer values),
- *RATIONAL* (the positive and negative rational numbers),
- *DURATION* (the duration, i.e. periods of time).

Basic types represent the most simple form of information used to describe the specified system. For example, a 'piston' is considered as a simple element that can not be described in term of more simple elements. Of course, this consideration is relative, and it depends on the wanted level of details in the produced specification.

2.2.2 Constructed Data Types

In this section, we find the declaration of the user defined types with a more complex structure. A constructed type is built using the following constructors:

- the Cartesian product (providing tuples) : *CP* ;
- the set : *SET* ;
- the bag or multi-set: *BAG* ;

- the sequence : *SEQ* ;
- the table (indexed bag) : *TABLE* ;
- the union (merging of values) : *UNION* ;
- the enumeration of values : *ENUM.*

A constructed data type is built from basic types, or from other constructed types. Here are some constructed data types :

BARE_BLOCK=CP[Id:BLOCK_ID,BearingsOn:BOOLEAN,Checked:ENUM[not,good,bad]]
BLOCK_PISTON=CP[Block:BARE_BLOCK,Pistons:SET[PISTON]]
BLOCK=UNION[BLOCK_STATE1,BLOCK_CRANKSHFT,BLOCK_VALVE,FINISHED_BLOCK]
BLOCK_IDS=SEQ[BLOCK_ID]
STATUS=ENUM[raw,processed]
BLOCK_STOCK=table[BLOCK_ID ->BLOCK]

We see that the first data type is constructed from basic data types (BLOCK_ID and BOOLEAN) and from another constructed data type (ENUM).

As basic data types, constructed types represent pieces of the information used to describe elements of specified system. For example, the second declaration might mean that a block of pistons is composed of a bare and some pistons fixed on it.

2.2.3 Operations

In this section, we specify the operations defined on data types. For example :

Removal :CAM_FOLLOWERS_PALLET x SET[CAM_FOLLOWER]->
CAM_FOLLOWERS_PALLET

An operation represents the fact of manipulating some elements to obtain some others. For example, the above declaration can have the following interpretation : removing some 'cam followers' from the 'cam followers pallet' results in a new 'cam followers pallet'.

2.2.4 Society Definition

In this section, we find the specification of the society composition : its agents and its sub-societies. In our example, we have :

SOCIETY Cell.Blocks_Transport_System
(AGV)))
(Buffer)

This declaration says that the society Blocks_Transport_System is a sub-society of the society Cell, and it is composed of several Agvs (the notation ')))' means more than one) and one Buffer (Figure 2.2[1]).

[1] The Albert II specification editor can generate this graphical representation of the specification.

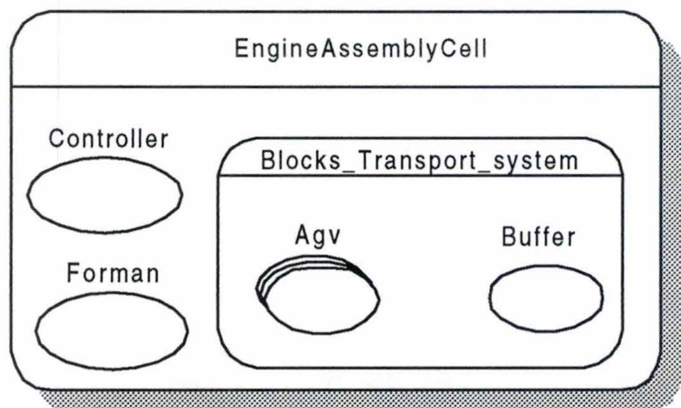

Figure 2.2: Structure of the Society Cell

2.2.5 Agent States Components

We find the declaration of an agent's state components under "STATE COMPONENTS" header. State components can be of four types :

- set component : set-of ;
- instance component : instance-of ;
- sequence component : sequence-of ;
- table component : table-of.

The value of a state component can be derived from the value(s) of other state component(s). This is declared by using the *derived-from* construct. Finally, state component can be private to his agent, or it can be accessed be other agent (the → construct). For example, the agent Buffer of the society Blocks_Transport_System has the following state components:

Content set-of BLOCK -> Cell.Controler
**Capacity* instance-of INTEGER -> Cell.Controler
BufferFull instance-of BOOLEAN derived-from Capacity, Content -> Cell.Controler

So, the state of the buffer is characterized by its *Content* of blocks, by its *Capacity* (the notation *Capacity means that is component has a constant value) and by the component *BufferFull*. The value of *BufferFull* is derived from those of *Content* and *Capacity*. And the three components can be seen by the agent *Controler* of the main society *Cell*.

2.2.6 Agent actions

The section "ACTIONS" regroup the declaration of the actions related to the agent. For example, the agent Controler of the society Cell performs one action :

RequestTransport(BLOCK) -> Cell.Blocks_Transport_System.AGV

This means that this agent requests the transportation of a block from the agent Agv. This later performs the following actions :

*Load(BLOCK)
*Transport(BLOCK)
Transportation

Some of these actions might be fictive, i.e. actions that are not related to real actions of the agent. For example, the action *Transportation* is specified just to be used later (see "ACTION COMPOSITION" section in Annex A) to regroup some actions that must be performed simultaneously (the notation **Load* means that this action must be performed in combination with some other actions).

The sections : data types, operations, state components and actions, are the only sections treated by our paraphraser. In fact, they are the declarative part of an Albert II specification. The can be used to paraphrase the remaining sections. So, to paraphrase completely a given specification, all want we need is precise the information provided by the analyst to paraphrase the declarative part.

We will see in the next chapter that the quality of the paraphrasing process output depends a lot on the quantity and the form of the provided information. Our objective was to ask a minimum of information concerning the treated part.

2.3 Non treated sections

Now, we will have a quick look at the remaining sections of an Albert II specification. For more details, see [Du97].

2.3.1 Derived components

We saw that the value of some state components can be derived from the values of some other components. In the 'Derived components' section, we precise the mathematical expression that defines the derived value.

2.3.2 Initial valuation

Some state components have initial values. These values are given in this section.

The sections 'Derived components' and 'Initial valuation' are called the 'Basic constrains' defined on an agent.

2.3.3 State behavior

In this section, we give the constrains that control the changes of the agent's state during the execution of an action, or during the complete life of the agent.

2.3.4 Actions composition

It is possible to consider as one action, many actions that must be executed together. In this case, we can precise the order of execution of these actions, and the delay between the termination of one action and the beginning of its successor.

Actions can be composed in many forms :

- Sequential composition: $a \leftrightarrow a_1, a_2, \ldots a_n$

The action a_1 starts at the same time as the action a, and the action a_{i+1} starts after the end of the action a_i, and the action a_n ends at the same time as the action a.

- Repetitive composition: $a \leftrightarrow \{a_1\}^n$

For each occurrence of a there is n successive occurrences of a_1.

- Parallel composition: $a \leftrightarrow a_1 || a_2, \ldots || a_n$

The action a starts at the same time as the action a_i starting the earliest, and ends at the same time as the action a_i ending the latest.

- Simultaneous composition: $a \leftrightarrow a_1 |\Leftrightarrow| a_2, \ldots |\Leftrightarrow| a_n$

All the actions a_i start and end at the same time as the action a does.

- Costarting composition: $a \leftrightarrow a_1 |\Rightarrow a_2, \ldots |\Rightarrow a_n$

All the actions a_i start at the same time as the action a does, and the action a ends at the same time as the action a_i ending the latest.

- Cofinishing composition: $a \leftrightarrow a_1 \Leftarrow| a_2, \ldots \Leftarrow| a_n$

The action a starts at the same time as the action a_i starting the earliest, and All the actions a_i end at the same time as the action a does.

- Alternative composition: $a \leftrightarrow a_1 \oplus a_2, \ldots \oplus a_n$

The execution of the action a corresponds to that of a_1 or a_2 ... or a_n. The action a_i starts and ends at the same time as the action a does.

2.3.5 Actions duration

In this section, we put constraints on the duration of the occurrences of actions.

The sections 'State behavior', 'Action composition' and 'Action duration' form the 'Declarative constraints' defined on an agent.

2.3.6 Actions precondition

Preconditions define the conditions that must be verified on an agent state so that a given action may occur in the next change of the state.

2.3.7 Effects of Action

These constraints are used to express the changes of an agent state du to the execution of an action. Such changes may be caused by the actions performed by the agent, or by the actions that he imports from other agents.

2.3.8 Triggerings

These constraints define conditions on the agent state under which a particular action has to be executed.

2.3.9 Actions perception

The constraints of this section define the behavior of an agent towards the actions imported from other agents of the society. There are three forms of action perception :

- Knowledge: K (external_Action / Condition)

Defines the external actions that an agent will be sensitive to, if a particular condition on the agent state is verified.

- Ignorance: I (external_Action / Condition)

Defines the external actions that an agent will ignore, if the condition on the agent state is verified.

- Exclusive knowledge: XK (external_Action / Condition)

Defines BOTH, the external actions that an agent will be sensitive to, if the condition on the agent state is verified, and the external actions that an agent will ignore, if the same condition on the agent state is not verified.

2.3.10 State perception

Beyond this heading we define how an agent sees parts of the state of other agents of the society that he can access. As action perception, an agent can know which parts he can see on certain conditions, which parts he can ignore, and which parts he can exclusively know (K, I, and XK).

2.3.11 Actions information

As an agent perceives the actions of other agents of his society, he also informs these agents about his own actions. Again, we find three ways of actions information : the actions that other agents know on certain conditions, those that they ignore, and those that they exclusively know (K, I, and XK).

2.3.12 State information

Beyond this heading we define how an agent shows parts of his state of other agents of the society that he can access. State information is also specified using the K, I, and XK connectives.

'Actions perception', 'State perception', 'Actions information' and 'State information' form together the 'Cooperative constraints' defined on an agent.

Remark

By examining the above non treated sections, we can easily see that there is a kind of repetition of information. For example, if an agent imports an action from another agent, this action will be declared in the section 'Actions perception' of the first agent and in the section 'Actions information of the second. So, to produce a good paraphrasing output when treating such sections, it would be a good idea to take in consideration the possible repetition of information.

2.4 Linguistic information

Now, we are going to precise the needed information that the analyst might provide, to enable the paraphraser to produce a good paraphrase of a given specification.

2.4.1 Necessity of linguistic information

The main objective of paraphrasing an Albert II specification, is to make this specification understandable by the customer, who generally is not familiar with formal languages. He presents his system's requirements using a language that he understands (English, French, etc.). The analyst specifies this system with Albert II (a language that he understands). The two sides must reach an agreement concerning the specification of the system. So it is preferable to present to the customer a document that he can understand (or we will ask our customers to learn the Albert II language).

The specification is a set of formal objects that the analyst chooses to simulate the real objects of the system: a 'piston' becomes a 'basic type', the 'Auto-Guided Vehicle' becomes the 'agent AGV', the 'activity of transporting a block' by the AGV becomes an 'action' of this agent, etc. So, having the formal specification, and knowing the formal semantics of the AlbertII objects, we already have a lot of information that can be reproduced in natural language. The problem is that these information might be unclear: we can reproduce the information 'the system has an agent called AGV', but what does 'AGV' means? To which real object is-it related? Questions that are not always evident to answer.

So, we need more information than that founded in the specification. For example, the analyst must explicit that the 'AGV' represents the 'Auto-Guided-Vehicle'. We have many classes of formal objects, and we tried to find what linguistic information we need to be able to reproduce a document understandable by the customer.

We indicated above (figure 2.1) the formal objects that we treat during the paraphrasing process. We tried to find the minimum of information needed to clarify the specification of these objects. We can regroup these objects in two groups: Those related to ENTITIES in the system and those related to ACTIVITIES of some of these entities. The needed linguistic information depends on the group of the treated object.

2.4.2 Formal objects related to ENTITIES

By entities, we mean material entities: 'piston', 'block', 'auto-guided-vehicle', 'worker', etc. or non material entities : 'block identifier', 'buffer capacity', etc. If fact, a way to find these objects is to detect the nouns or noun phrases (from a linguistic point of view) in the treated specification.

In the specification, these entities are: data types, agents, societies or state components.

When we see one of these formal objects, we can suppose that there are three possibilities concerning the comprehension of its real semantic, and then relating it to a real entity :

a) Either we understand directly what does this object represent. For example the basic data type 'piston'. In this case we **don't need any new information**.
b) Or we will need just to explicit the name of the object to understand its signification. For example the agent 'AGV' needs just to precise that it represents the 'Auto-Guided-Vehicle',

and the customer, who is familiar with the system, will understand the semantic of 'AGV'. Here, we need to provide the **significant name** of the object.

c) Or, even the new name is not enough to explicit completely the real semantic of the object. In this case, we estimate that the analyst must provide what we called the **denotation** of this formal object. For example, the basic data type 'block_id' may be considered as not representative, so the analyst will add that he uses this object to represent the 'block identifier'. If the identifier is a 'sequence of characters that must starts by a capital letter', the name will not be sufficient. The denotation is must useful when the analyst defines objects that are not explicitly present in the real system, or when he defines real objects with a totally or partially non conventional semantic.

We can imagine many situations similar to that of the analyst writing a formal specification. For example, when writing a text about the education in Belgium, the author mentions the word 'FUNDP'. If he wants to be sure that the reader will understand the text, he provides that 'FUNDP' is the abbreviation of 'Facultés Universitaires Notre-Dame de la Paix'. If he wants to be precise about this term, he may add that 'FUNDP' is a 'private university at Namur'.

So, our linguistic information are: an optional new name and an optional denotation of the formal objects. In addition, for some objects we need the **plural name**, because we use this linguistic information to paraphrase objects that use this formal object(Set of, sequence of, etc.). And finally, giving the **article** ('the', 'a' or 'an') of the object is welcomed to ameliorate the quality of the paraphraser output. We will say more about these two later information when speaking about the paraphrasing templates.

2.4.3 Formal objects related to ACTIVITIES

In this group, we have the actions of agents and the operations on data types. We considered that the operations can be seen as global actions that can be performed by any agent.

From the formal specification of an operation, we can give an acceptable semantic, because any operation is the act of manipulating some objects to modify them or to produce new objects. For example, let's consider the operation :

Removal:CAM_FOLLOWERS_PALLET x SET[CAM_FOLLOWER] →
CAM_FOLLOWERS_PALLET

By default, we simply paraphrase it by 'Removal : <u>determines</u> a CAM FOLLOWERS PALLET from a CAM FOLLOWERS PALLET and a set of CAM FOLLOWERS', but it will be better if the analyst gives the real effect of the operation : for example, he gives the description 'Removal: <u>remove</u> a set of CAM FOLLOWER from a CAM FOLLOWERS PALLET'. Because we use the template-based techniques, there is no magical way to allow us to interpret the operation as in the second form.

In the case of an action, the description of the action is very necessary to well give the real semantic of the action. The action name is not always enough to give its semantic. An action may have arguments that can be fixed (input to the action), or modified during the execution of the action(output or input/output), and it is very difficult to distinguish between the two kinds. So, the simple interpretation action is less significant that of an operation. For example, the agent Controler performs the action :

RequestTransport(BLOCK) -> Cell.Blocks_Transport_System.AGV

In a traditional template-based way, this action can be interpreted as 'RequestTransport the Block to the AGV' or 'RequestTransport the Block by the AGV', but it must be interpreted as 'request the transportation of a block by the AGV'. Simple, and very similar to the second interpretation, but difficult to find without the intervention of the analyst.

So, for these two kinds of Albert II objects, we estimate that we need what we called the **description** of the action or the operation.

2.4.4 Linguistic precessions

To produce a compatible paraphrase of the treated specification, the provided information are supposed respecting the following linguistic forms:

- the name, the plural name and the denotation of an object: provided under **noun phrases** forms without any article.
- by default, with no plural name provided, the plural name is the name+'s' (or 'es' or 'ies').
- the description of an action or an operation : provided as any sentence under the form **verb+complement**.

In fact, the resulting paraphrase of an object is directly monitored on the screen, so the analyst can adjust the provided information according to what he sees.

4.4.5 Providing linguistic information

We will see in the chapter five how the analyst can provide the linguistic information for objects in an Albert II specification.

Chapter Three

The Paraphrasing Patterns

Chapter Three

The paraphrasing patterns

In this chapter, we are going to detail our paraphraser output. We said that it uses template-based techniques, so first we will present individual paraphrases for each kind of AlbertII formal object, then we speak about the global paraphrase.

3.1 individual paraphrasing patterns

Next, we are going to give the paraphrasing pattern for each Class of the AlbertII objects, and we will give the expected paraphrase for some of this objects.

3.1.1 Conventions

- A_x denotes the linguistic x to be provided by the analyst (e.g. : A_name means the name given by the analyst).
- Plu_x denotes the plural of x.
- F_x denotes the formal x as it appears in the specifications (e.g. : F_name).
- Para_x denotes the paraphrase of x.
- In the given examples, the information supposed to be provided by the analyst will be underlined.
- As our paraphraser produces two forms of paraphrases, we are going to give these two forms (Analyst-oriented and Customer-oriented) under the header 'Output'.

3.1.2 Basic data type

Syntax F_TypeName

Output
- Analyst: *F_TypeName.*
- Customer: *A_name* : *A_Definition* .

3.1.3 Redefined data type

Syntax F_RedefTypeName = F_TypeName or
F_RedefTypeName = F_TypeName* (extending the type with the value Undef)

Output
- Analyst: *F_RedefTypeName*: a variable of this type can have a value of type *F_TypeName* (or the special value Undef).

- Customer: *A_RedefTypeName: A_TypeName* or nothing.

3.1.4 Cartesian Product data type

Syntax F_CpTypeName = CP[$Field_1$:$TypeExpr_1$, $Field_2$:$TypeExpr_2$,..., $Field_n$:$TypeExpr_n$]

Output
- Analyst : *F_CpTypeName*: tuple composed of the field $Field_1$ of type $TypeExpr_1$, the field

Field$_2$ of type TypeExpr$_2$ ... and the field *Field$_n$* of type *TypeExpr$_n$*.

- Customer: *A_CpTypeName*: *A_SetTypeDef*. It is a tuple composed of *A_Field$_1$Name*, *A_Field$_2$ Name*, ... and *A_Field$_n$Name*.

the word 'tuple' is replaced by 'couple' or 'triple' or 'quadruple' if the number of the fields is 2 or 3 or 4. Normally, this simple text analysis is not done in the template-based techniques, but we can do such analysis because we have enough information about the input specification (the number of fields of the main object).

3.1.5 Set/seq/bag data type

Syntax F_SetTypeName = SET [TypeExpr]

Output
- Analyst : *F_SetTypeName*:set/sequence/bag of values of type *TypeExpr*.
- Customer: *A_SetTypeName: A_SetTypeDef*. it is a set/sequence/multi-set of *Plu_TypeExpr*.

3.1.6 Table data type

Syntax F_TabTypeName = TABLE [TypeExpr1 → TypeExpr2]

Output
- Analyst: *F_TabTypeName*: table of values of type *TypeExpr2* and the indexed by values of type *TypeExpr1*.

- Customer: *A_TabTypeName*: *A_TabTypeDef*. It is a list of couples. Each couple is composed of *A_TypeExpr1Name* and the corresponding *A_TypeExpr2Name*.

Example : STOCK = TABLE [BLOCK_ID -> BLOCK]

Output
- Analyst: STOCK: table of values of type BLOCK and the indexed by values of type BLOCK_ID.
- Customer: *Stock*: Stock of blocks. It is a list of couples. Each couple is composed of a block identifier and the corresponding block.

3.1.7 Union data type

Syntax F_UnTypeName = UNION [TypeExpr$_1$, TypeExpr$_2$,...TypeExpr$_n$]

Output
- Analyst: *F_UnTypeName*: a value either of *Para_TypeExpr$_1$* or of *Para_TypeExpr$_2$* ... or of *Para_TypeExpr$_n$*.

- Customer: *A_UnTypeName*: *A_UnTypeDef*. A *A_UnTypeName* is *A_TypeExpr$_1$Name* or *A_TypeExpr$_2$Name*... or *A_TypeExpr$_n$Name*.

3.1.8 Enumerated data type

Syntax F_EnumTypeName = ENUM [Const$_1$, Const$_2$,..., Const$_n$]

Output

- Analyst: *F_EnumTypeName*: It is defined as one of the following constant values : $Const_1$, $Const_2$,... and $Const_n$.

- Customer: *A_EnumTypeName*: *A_EnumTypeDef*. A *A_EnumTypeName* is $Const_1$ or $Const_2$ ... or $Const_n$.

Remarks on paraphrasing data types

- Two main differences between analyst-oriented output and customer-oriented one. The first is that for the customer, we tried to avoid as possible the use of formal terms (type, index, union, enumerated, etc.). The second is when we paraphrase a constructed type with another constructed type inside it. In the case of analyst output, we paraphrase the interior data type, whereas in the other case, we give the analyst the choice between using its name or paraphrasing it.
- Sometimes, we can find data types defined only to be used in the specification, so they have no corresponding objects in the system. These data types can be ignored when producing customer-oriented output (when the analyst doesn't deselect the autmatic and the manual paraphrase).
- We will see later, when presenting the paraphrasing output formats, that basic and constructed types are grouped together as the DICTIONARY of the terms used in the specification. In fact, these objects are defined to make the specification simpler and clearer. For example, the analyst could specify the object 'BLOCK_STOCK' as

BLOCK_STOCK=table[BLOCK_ID ->
UNION[CP[Block:BLOCK_PISTON,Crankshafts:SET[CRANKSHAFT],ScrewedOk:BOOLEAN],
BLOCK_VALVE,FINISHED_BLOCK]]

instead of

BLOCK_STOCK=table[BLOCK_ID ->BLOCK]
BLOCK=UNION[BLOCK_CRANKSHFT,BLOCK_VALVE,FINISHED_BLOCK]
BLOCK_CRANKSHAFT=CP[Block:BLOCK_PISTON,Crank:SET[CRANKSHAFT],ScrewedOk:
BOOLAN]

Data types can be seen as terms or definitions or conventions used in the specification of the system, so we put them in what we called DICTIONARY (of terms).

3.1.9 Operations

Syntax F_OpName : $TypeName_1$ x $TypeName_2$...x $TypeName_n$ $\rightarrow$ TypeName

Output

- Analyst : *F_OpName*: determine a value of type *TypeName* from values of types: $TypeName_1$, $TypeName_2$ ... and $TypeName_n$.

- Customer: *F_OpName*: determine *A_TypeName* from $A_TypeName_1$, $A_TypeName_2$ ...and $A_TypeName_n$.

In the customer-oriented output, we gave the default output because normally, the analyst is encouraged to provide a description of the operation.

Example:

Removal:CAM_FOLLOWERF_PALLET x SET[CAM_FOLLOWER] -
>CAM_FOLLOWERS_PALLET

Output

- Analyst : *Removal*: determine a value of type CAM_FOLLOWERS_PALLET from values of types: CAM_FOLLOWERS_PALLET and SET[CAM_FOLLOWER].
- Customer: *Removal*: determine a followers pallet from a set of followers and a followers pallet.

Or (provided by the analyst)

Removal: remove a set of cam followers from a cam followers pallet.

3.1.10 Society composition

An example will be enough to show the way of paraphrasing society objects.

SOCIETY Cell.Blocks_Transport_System
(AGV)))
(Buffer)

Output

- Analyst : Blocks_Transport_System : sub-society of the society Cell. It is composed of several AGV agents and one Buffer agent.
- Customer : Blocks transport system : sub-society of the society engine assembly cell. It is composed of several auto-guided vehicles and one buffer.

3.1.11 Set/Sequence State component

Syntax F_CompName **set-of** F_TypeName
F_CompName **sequence-of** F_TypeName

Output

- Analyst: *F_tCompName*: set/sequence of values of type *F_TypeName*.
- Customer: *A_CompName*: *A_CompDef*. It is defined as a set/sequence of *Plu_TypeName*.

3.1.12 Table State component

Syntax F_CompName **table-of** F_TypeName$_1$ **indexed-by** F_TypeName$_2$

Output

- Analyst : *F_CompName*: table of values of type *F_TypeName*$_1$ and indexed by values of type *F_TypeName*$_2$.
- Customer : *A_TabComName*: *A_TabComDef*. It is defined as a list of couples. Each couple is composed of *A_TypeName*$_1$ and the corresponding *A_TypeName*$_2$.

3.1.13 Instance State component

Syntax F_CompName **instance-of** F_TypeName

Output
- Analyst: *F_CompName*: a single value of type *F_TypeName.*
- Customer: *A_CompName*: *A_tCompDef.* It is *A_TypeName.*

When we produce a customer-oriented paraphrase, if we encounter a predefined type of AlbertII (see 2.2.1), and if a linguistic name is not provided, we replace it with its real semantic:

- INTEGER, REAL, RATIONAL : number.
- BOOLEAN : true or false.
- CHAR : character.
- STRING : sequence of characters.

3.1.14 Derived component

We saw that values of some state components can be derived from values of others. In this case, we mention this when paraphrasing derived components.

Syntax **derived-from** $F_CompName_1$, $F_CompName_2$,... $F_CompName_n$

Output
- Analyst: The value of this component is derived from the value (s) of the component (s) $F_CompName_1$, $F_CompName_2$,... and $F_CompName_n$.

- Customer: it is derived from $A_CompName_1$, $A_CompName_2$,... and $A_CompName_n$.

3.1.15 Exported component

A state component of an agent can be exported to other agents ($Agent_1$, ...,$Agent_n$). This information is added to the paraphrase of this state component:

CompName can be perceived by the agent(s) : $Agent_1$, $Agent_2$... *and* $Agent_n$.

3.1.16 Constant component

A state component of an agent can have a constant value. This is defined formally by preceding the component name by the character '*'.

Output: this component has a constant value.

Example:

BufferFull instance-of BOOLEAN derived-from Capacity,Content → Cell.Controler,Cell.Forman

Output (Customer form)

bufferfull: it is true or false. It is derived from the capacity and the content, and it can be perceived by the controler and the Forman.

Remark: an instance state component of type BOOLEAN could by seen -from a linguistic point of view- as an adjective of his agent. We did not consider this case because we estimated that is not frequent.

3.1.17 Actions

An action is paraphrased according to the description provided by the analyst. When this description is not provided, we just give the formal definition of the action.

3.1.18 Exported action

Like a state component, an action can be exported to other agents. This information is added to the action paraphrase.

3.1.19 Combined actions

Some actions must always be executed in combination with some others. This information is also mentioned in the action paraphrase.

Example: the agent Controler performs the action

*RequestTransport(BLOCK) -> Cell.Blocks_Transport_System.AGV

The analyst provides the description: 'request the transportation of BLOCK'.

Output : RequestTransport: request the transportation of a block. This action is always performed in combination with some other actions, and it is perceived by the auto-guided vehicle.

3.2 Global paraphrasing output

The global paraphrase of the treated specification is organized as follows :

1. Basic and constructed data types and operations are grouped together as the DICTIONARY of the terms used in the specification. For every object, we have the paraphrase of the object followed by its formal specification (if the analyst wants to keep it).
2. Then the paraphrase of the composition of main society. Just the paraphrase: ' the society is composed of ...'.
3. Then the paraphrase of the agents and sub-societies of the main society. For an agent, we paraphrase his state components, then his actions. For a sub-society, we repeat the above steps 2.3, because here there is no dictionary.

Chapter Four

Parsing Basics

Chapter Four

Parsing Basics

A parser of a language is a program that takes as input a string *w* written in this language, and produces either a *parse tree* for *w*, if *w* is a correct sentence, or an error message indicating that *w* is not a sentence of the language. A parse tree is a diagram which exhibits the syntactic structure of the treated string (figure 4.1). Often the parse tree is produced in only a figurative sense because in reality, it exists only as a sequence of actions made by stepping through the tree construction process.

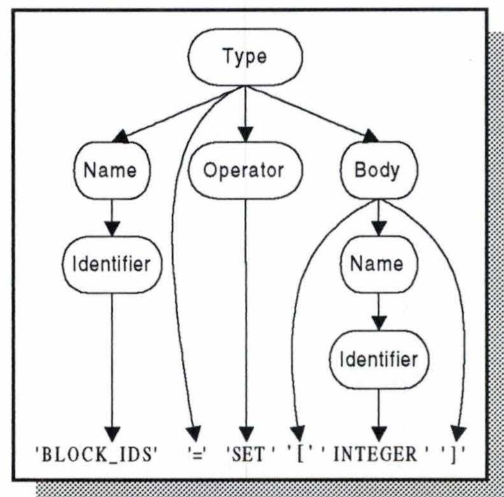

Figure 4.1 : A parse tree of BLOCK_IDS = SET [INTEGER]

Parsing a string is composed of two phases. The first phase, called the *lexical analyzer*, or scanner, separates characters of the input string into groups that logically belong together ; these groups are called *tokens*. The usual tokens are keywords of the language, such as SET or derived-from[1], identifiers, such as X or Block, operator symbols such as + or =. The output of the lexical analyzer is a stream of tokens, which is passed to the next phase, the *syntax analyzer*. The lexical analyzer groups tokens into syntactic structures. For example, the three tokens representing A+B might be grouped into a syntactic structure called an *expression*. Expressions might further be combined to form statements. Often the syntactic structure can be regarded as a tree whose leaves are the tokens. The interior nodes of the tree represent strings of tokens that logically belongs together.

4.1 Lexical analysis

The lexical analyzer is the interface between the source text and the parser. The lexical analyzer reads a source text one character at a time, and separating this suite into a succession of atomic units called *tokens*. Each token represents a sequence of characters that can be treated as a single logical entity. Identifiers, keywords, punctuation symbols such as commas and parentheses, operators are typical tokens. For example, In the AlbertII statement :

VALVES = SET[VALVE]

[1] Keywords of AlbertII language

we find the following tokens: VALVES; =; SET; [; VALVE;].

What is called a token depends on the language of the treated text.

4.1.1 Regular expressions

the concept of *regular expression,* used to specify the tokens of a language. Lets first produce a few basic terms dealing with languages.

- *Alphabet or character class*: denotes any finite set of symbols. The terms *symbol* and *character* can be used synonymously. For example: the set {0,1} is an alphabet. It consists of two symbols, 0 and 1, and it is often called *binary alphabet.* Another example of alphabet is the well known ASCII character set.
- *String*: denotes any finite sequence of symbols, such as 001. A sentence is a string.
- *Language*: denotes any set of strings formed from some specific alphabet.

We said that the lexical analyzer transforms the input string into tokens. A token is either a single string (such as a punctuation symbol) or one of a collection of strings of a certain type (such as an identifier). If we view the set of strings in each token class as a language, we can use the regular expression notation to described tokens.

For example, in many languages, an identifier is defined as a letter followed by zero or more letters or digits. We can say this in English, but it is rather difficult to construct automatically a lexical analyzer from this description.

Using regular expression notation, we could write

identifier = letter (letter | digit)*

From a description of this nature we can automatically construct a program that recognizes identifier.

4.2 Syntax analysis

The lexical analyzer and the following phase, syntax analyzer, are often grouped together into the same pass. The lexical analyzer operates either under the control of the syntax analyzer or as a coroutine with it. The syntax analyzer asks the lexical analyzer for the next token, whenever the parser needs one.

The syntax analyzer has two functions. The first is the detection of the syntax errors in the tokens appearing in its input, which is the output of the lexical analyzer. It checks which of these tokens occur in patterns that are permitted by the specification for the source language. For example, if an Albert II specification contains the expression

X =SET [[Y]

then after lexical analysis this expression might appear to the syntax analyzer as the token sequence

identifier = set_operator [[identifier]

On seeing the second [, the syntax analyzer should detect an error, because the second [violates the syntax of a Set type in Albert II language.

The second function of the syntax analyzer is to make explicit the hierarchical structure of the incoming token stream by identifying which parts of this stream should be grouped together. For example, the expression

A / B *C

has two possible interpretations:
1. divide A by B and then multiply by C; or
2. multiply B by C and then use the result to divide A by the result of multiplication.

The language specification must tell us which interpretation is to be used. This is done by giving the *precedence* and the *associativity* of operators. We will see this notion when we speak about Visual Parse++.

As the lexical analysis is based on the notion of regular expressions used to describe tokens, the syntactic specification of a language is done using the notion of grammars, and in particular, the notion of *context-free grammars*, which is also sometimes called a BNF (Backus-Naur Form) description. Using this notation to describe the syntax has many advantages.

- A grammar gives a precise syntactic specification of the language.
- An efficient parser can be constructed automatically from a properly designed grammar.
- A grammar imparts a structure to a program that is useful for detection of errors.

4.2.1 Context-free grammars

In general, a grammar involves four quantities:

Terminals : are the basic symbols composing stings of the language. The word *token* can be considered synonym for *terminal.* For example, the strings 'CP', 'Valve', 'BLOC_ID' and '=' are terminals.

Nonterminals: are special symbols that denote sets of strings, or syntactic categories. For example, to specify the list of constructed data types in Albert II language, we state that :

"A constructed data type is defined as an identifier, followed by '=', followed by the body of the type. The list of constructed data type is formed of one or more constructed data type".

'Constructed data type' and 'list of constructed data types' are syntactic categories. One nonterminal is selected as the *start symbol,* and it denotes the specified language. The other nonterminals are used to define other sets of strings that help defining the language.

Productions: define the ways in which the syntactic categories may be built up from one another and from terminals. For example, the above statement about the list of constructed data types, can be defined by the following productions:

DataType → identifier '=' body
ListTypes → DataType
ListTypes → DataType ListTypes

4.3 Basic parsing techniques

Defining the tokens and the syntax of a language is not sufficient to build a parser for this language. We must also precise how to check whether an input string is a sentence of the grammar and how to construct a parse tree from the string.

The two basic types of parsers for context-free grammar are the bottom-up and the top-down parsers. Bottom-up parsers build parse tree from the bottom (leaves) to the top (root), while top-down parsers start with the root and work down to the leaves. In both cases, the input to the parser is being scanned from left to right, one symbol at a time.

Here, we will present only the technique of bottom-up parsing because it is the technique used by Visual Parse++, the program that we uses to build our parser.

The bottom-up method is called « **shift-reduce** » parsing because it consists of shifting input symbols onto a stack until the right side of a production appears of top of the stack. The right side may then be reduced to the symbol on the left side of the production.

For example, consider the grammar defining the syntax of basic types in Albert II language

BTypeList -> BasicType
BTypeList -> BasicType BTypeList
BasicType -> Identifier {Identifier : represents any character string}

and the input string *w* = PISTON BLOCK_ID. We want to reduce this string to the starting symbol BTypeList. This can be done as follow:

1. We scan *w* looking for substrings that match the right side of some production. After scanning the whole substring PISTON we find that the third production matches (PISTON is an identifier), so we replace PISTON by Identifier and we have the production BasicType **->** Identifier that matches the substring PISTON. We obtain *w* = BasicType BLOCK_ID.
2. We do the same thing with BLOCK_ID which can be reduced to BasicType and so *w* = BasicType BasicType.
3. Then we reduce the last BasicType of *w* to the nonterminal BTypeList because we have the production BTypeList **->** BasicType, so we obtain *w* = BasicType BtypeList.
4. Now we have the production BTypeList **->** BasicType BTypeList that matches the entire string w so, we reduce *w* to the right side of this production to obtain *w* = BTypeList.

We said that the output of the syntactic analyzer is a parse tree of the treated string. We can consider that the sequence of the productions used in some derivation is an example of an *implicit representation.* A linked list structure for the parse tree is an *explicit representation.*

When we use the Shift-Reduce parsing technique, two kinds of conflicts may appear in our grammar. A *Shift/Reduce* conflict appears when we define two (or more) productions that have common prefix on their right side. For example, in Pascal programming language, we have the productions:

Statement -> **if** Condition **then** Statement
Statement -> **if** Condition **then** Statement1 **else** Statement2

A Shift/Reduce conflict will arise if we have an input string containing the **else** clause. After recognizing the **then** statement, the parser is not able to decide wither to <u>reduce</u> it by applying the first production, or to continue (to <u>shift</u>).

The second type of conflicts is the *Reduce/Reduce* conflict that might appear if the grammar contains two productions that can correspond to the same input. For example, the two productions

Statement -> Identifier
Statement -> 'BasicTypes'

will cause a Reduce/Reduce conflict for the input string *w* = 'BasicTypes A B C'.

We will see later how Visual Parse++ resolves these conflicts if it detects them in the defined grammar.

4.4 Building a parser with Visual Parse++

Visual Parse++ is the program that we uses to define our lexical and syntax analyzers of our parser. The definition of the lexical analyzer is done in the %expression section(s) where we give the list of the regular expression describing the token of the language. The syntactic analyzer is defined in the %prec section and the %production section. The %prec section contains precedence information used to resolve conflicts in the following grammar, and the %production section defines the parser in a BNF-like notation. The sections must be specified in the order listed.

4.4.1 The Lexical analyzer in Visual Parse++

The lexical analyzer is defined in the %expression section(s) of the rule file. Each %expression section defines a named regular expression list. Visual Parse++ maintains a stack of %expression lists that you can %push, %goto, and %pop as regular expressions are recognized. This makes it easy to do some context dependent processing in the lexical analysis phase, which can greatly simplify the grammar design. For instance, it is trivial to handle things like comments or quoted strings, which can be difficult to design into a grammar.

The first expression list encountered in a rule file is the active lexical analyzer when lexing begins. The %expression statement has syntax:

%expression name

Each line in an expression list has syntax:

'regexpr' nameOrIgnore [action];

'regexpr' A regular expression. Regular expressions are described below.
nameOrIgnore Either a valid name or the reserved word %ignore. %ignore tells the lexical analyzer
to recognize but ignore the token. The token is not passed to the parser. The name or %ignore is required. The name is used to generate symbols used in the language bindings to identify this regular expression.
action The optional action can be %push name, %goto name, or %pop. %push pushes the named expression list on the stack. This list becomes the active expression list.
%pop pops the current expression list off the stack. %goto is equivalent to a %pop followed by a %push.

If two (or more) regular expressions recognize the same language, the ambiguity is resolved in favor of the one that occurs later in the specification.

In Visual Parse++, regular expressions are defined as patterns composed of normal characters and meta-characters, which have special meaning. The normal characters are any character. The meta-characters are:

. Matches any single character except newline (\n).

* A postfix operator that matches 0 or more copies of the preceding regular expression.

[ ] Matches any character enclosed in the brackets. The term for this construct is character class. A range of characters is indicated with a '-' (0-9, a-z, etc.). If the first character is a '^', the meaning is to match any character not in the class.

+ A postfix operator that matches 1 or more copies of the preceding regular expression.

( ) Groups a series of regular expressions.

^ As the first character of a regular expression, matches the beginning of a line. The regular expression will not be recognized unless it is anchored at the start of a line. The start of line character is not considered part of the recognized lexeme.

$ As the last character of a regular expression, matches the end of a line. The regular expression will not be recognized unless it is immediately followed by a newline (\n). The newline character is not considered part of the recognized lexeme.

" " A literal match. Turns off recognition of all meta-characters except the backslash (\) sequences.

| The or operator. An infix operator that will match either the left regular expression or the right regular expression.

? A postfix operator that matches 0 or 1 occurrences of the preceding regular expression.

{m,n} A postfix operator matching between m and n occurrences of the preceding regular expressions.

{name} Macro substitution.

\ The meaning depends on what follows. The following characters have special meaning.

a	BEL
b	backspace
f	formfeed
n	newline
r	carriage return
t	tab
v	vertical tab
d[0-9]+	decimal number

o[0-7]+	octal number
x[0-9a-fA-F]+	hexadecimal number
Other	The value of the character. This is useful for turning off the special meaning of meta-characters.

Lets illustrate some of these features by some tokens used in our lexical analyzer.

'[A-Za-z][A-Za-z0-9_]* | [A-Za-z][A-Za-z0-9_]**'	: Identifier.
'SET|[sS]et'	: keyword 'Set'.
'CP|cp'	: keyword 'CP'.
'[0-9]+ '	: integer numbers >= 0.

The backslash (\) in the first regular expression is required because '*' is a meta-character with a special meaning in a regular expression. In our case, we use the token identifier to recognize words like 'Controler', or words like 'Integer*'. The '\' turns off the special meaning of the meta-character '*'.

Rule files are platform and language (in the programming sense) independent. Visual Parse++ generates multiple language bindings based on one rule file. The supported languages are : C, C++, Delphi and Visual Basic.

4.4.2 Syntax analyzer in Visual Parse++

The syntax analyzer is defined in %prec and %production sections.

%prec

The precedence section is used to resolve any shift/reduce conflicts present in the grammar. For applications that contain things like arithmetic expressions, it is sometimes desirable to design the grammar knowing that conflicts are present. The grammar is more readable, and the conflicts can be removed by providing precedence information in the %prec section. Conflicts are discussed later. The syntax is:

%prec number nameOrAlias associativity

number	The precedence, which must be a number.
nameOrAlias	The name or 'alias' of an entry in an expression list.
associativity	The associativity of the token, either %left, %right, or %nonassoc.

Example: in a grammar defined to parse arithmetic expressions, we have

%prec

1, '+', %left;
1, '-', %left;
2, '*', %left;
2, '/', %left;

meaning that the operations '+' and '-' have the same precedence and that they have priority on the operations '*' and '/'. So the expression A+B*C-D will be interpreted as (A+B)*(C-D).

%production

The %production section defines a grammar in a BNF-like notation. Grammars are made of production rules, which the parser uses to recognize valid input. Production rules are made of terminals and nonterminals. Terminals are names or 'aliases' defined in %expression lists. Nonterminals are any symbol that is not a terminal.

The production statement looks like:

%production *startSymbol*

startSymbol is a nonterminal used as the starting symbol for the grammar.

Each rule has the form: label leftside -> [rightside] [%prec alias];

label	A production label used by the language bindings to identify the production.
leftside	A nonterminal symbol.
rightside	The rightside is a serie of zero or more of the following:

- A terminal symbol, i.e., a name or 'alias' of an entry in a regular expression list.
- A nonterminal symbol.
- A production with zero symbols on the rightside is a null production.

%prec alias	The optional precedence assigned to the production. A production has the same precedence as the rightmost terminal symbol in the production, unless overridden by this parameter. If the production has no terminal symbol and no %prec parameter, the precedence is 0. The *alias* specified on an entry in an regular expression list.

Examples of productions from the grammar of our parser:

// Staring symbol

Start	spec	-> SpecStart Identifier TypesOpsDec RootSoc;

// Basic Types

BasicDec	BasicDec	-> 'BASIC TYPES' BTypeList;
BType	BTypeList	-> BasicType;
BTypeList	BTypeList	-> BasicType BTypeList;
BasicTypeNo	BasicType	-> Identifier;

Shift/Reduce Conflicts

The %prec section is used to resolve the possible Shift/Reduce conflicts in the defined grammar. The precedence of the shift token is compared to the precedence of the production. Productions have the same precedence as the rightmost terminal symbol in the production. Productions have precedence zero if they don't contain a terminal symbol. Tokens have precedence zero unless they are included in the %prec section. The following rules resolve the conflict:

- If the token precedence is greater than the production precedence, the shift is taken.
- If the production precedence is greater than the token precedence, the reduce is taken.
- If the precedence values are equal and the token is %left associative (the default) the reduce is taken.
- Otherwise, the shift is taken.

Example : a simple arithmetic expression grammar like the following:

```
%expression

'[ \t\n]+'      %ignore;
'\+'            Plus,    '+';
'\*'            Mult,    '*';
'\-'            Minus,  '-';
'/'             Divide,  '/';
'[0-9]+'        Num,    'n';

%prec

1, '+',         %left;
1, '-',         %left;
2, '*',         %left;
2, '/',         %left;

%production expr

ExprPlus        expr -> expr '+' expr;
ExprMult        expr -> expr '*' expr;
ExprMinus       expr -> expr '-' expr;
ExprDivide      expr -> expr '/' expr;
```

Without the %prec section, this grammar would generate numerous shift/reduce conflicts.

Reduce/Reduce conflicts

For unresolved reduce/reduce conflicts, precedence is given to the production that occurs first in the grammar.

The complete grammar of our parser is in annex B.

Chapter Five

The Paraphraser

Chapter Five

The AlbertII paraphraser

This chapter is devoted to describe the method followed to realize our paraphraser. We try to follow the main steps, and to use some of the concepts exposed in [Dubois97].

5.1 The paraphraser requirements

The first step in the construction of our paraphraser is the analysis of the paraphraser requirements provided by the analyst who will use the paraphraser.

Our task is to implement a program which allows to paraphrase a part of a specification written in Albert II specification language. The treated part concerns the objects: data types, operations, agents, agents state components and agents actions.

The program must enables the analyst to:

- analyze of the specification and to display the list of the detected objects of the treated classes. The input specification is supposed syntactically correct, and having a predefined organization.
- provide the needed linguistic information concerning the detected objects.
- generate the paraphrase of these objects.
- save and modify the provided linguistic information.
- import linguistic information from an old paraphrase.

5.2 The Paraphrasing process

After analyzing the paraphraser requirements, we can imagine the scenario of the paraphrasing process. There are two possible scenarios as shown in figure 5.1.

In the first scenario, the paraphrasing process starts by opening the input file containing an Albert II specification that will be paraphrased for the first time. This file is normally generated by the Albert II editor. The specification (supposed syntactically correct) is analyzed (parsed) to recognize the Albert II objects of the treated classes and the parse tree is built during this analysis.

In the second possible scenario, the specification was already parsed and the analyst has provided some or all the linguistic information, and he wants to modify them, or to generate the corresponding paraphrase. In this case, the paraphrasing process starts by locating the text file containing the specification. This specification is parsed to build the parse tree of the specification, then we add the old linguistic information to the objects of the parse tree.

After its construction, the parse tree is monitored to the analyst to enable him to perform the desired actions: select an object to see, provide or modify the corresponding linguistic information, import information from another paraphrased specification, generate the paraphrase of the current specification, save the linguistic information, change the current dictionary, and finally terminate the paraphrasing process.

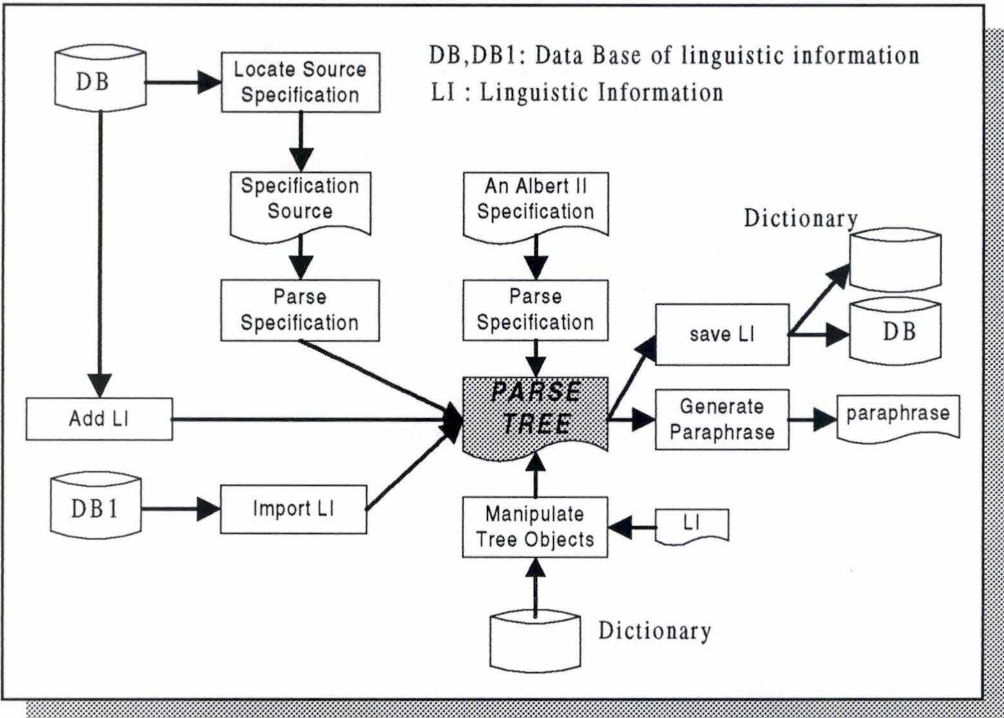

Figure 5.1 : Scenarios of the paraphrasing process

5.3 The Logical architecture

The logical architecture of a program is the set of its units defined in an abstract way without considering the programming environment that will be used to implement the program.

In our case, the logical architecture is the one shown in figure 5.2.

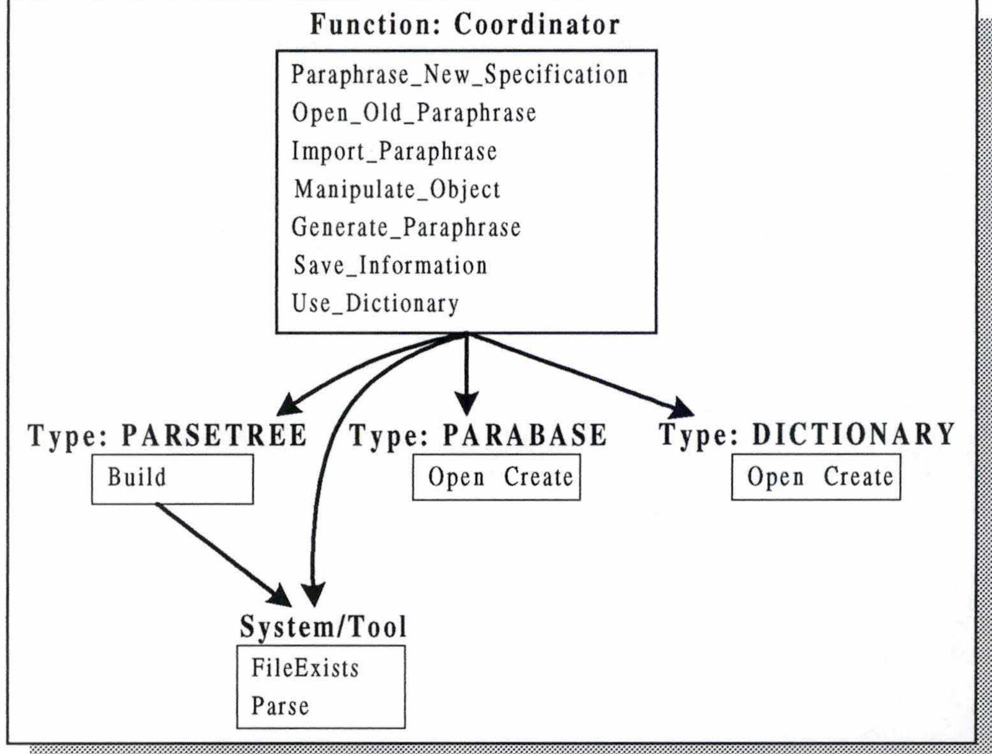

Figure 5.2: the logical architecture of the paraphraser

Next, we will specify the units of this architecture. **It is worth to note that our objective is to show the main ideas behind our paraphraser, and not to provide a rigorous specification,** so the use of a mixture of natural language, formal and programming concepts.

5.3.1 Function COORDINATOR

Interface

Paraphrase_New_Specification: STRING → PARSETREE
Open_Old_Paraphrase: STRING → PARSETREE
Import_Paraphrase: PARSETREE,STRING,BOOL,BOOL,BOOL → PARSETREE
Generate_Paraphrase: PARSETREE,STRING,FORMAT → SET[TEXTFILE]
Save_Information: PARSETREE,STRING,DICTIONARY →PARABASE,DICTIONARY
Use_Dictionary: STRING → SET [STRING]
Manipulate_Object: OBJECT,DICTIONARY → OBJECT

OBJECT = UNION [CTypeObject,BTypeObject, OperationObject, SocietyObject, AgentObject,CompObject, ActionObject]
TEXTFILE = SEQ [CHAR]
FORMAT = [Text, Html]

Specification

- Ptree = **Paraphrase_New_Specification** (FileName)

Precondition

FileExists(FileName+'.txt') and it contains an Albert II specification.

Postcondition

Ptree.Name = FileName ;
Ptree.Data =Build (FileName) ; | Parse the specification found in the file, and build the Parse tree during the parsing process.
Show (Ptree); | Enable the analyst to see the content of the parse tree.

- Ptree = **Open_Old_Paraphrase** (BaseName)

Precondition

Pbase = Open (BaseName) and FileExists(Pbase.Data[1].Name+'.txt').

| In fact, we don't save the structure of the specification when saving the provided linguistic information, so we will need to reparse the specification to find its structure, then we add the saved information. The path to the specification is saved in the first record with the linguistic information.

Post Condition

Pbase = Open (BaseName) ;
SpecName = Pbase.Data[1].Name;
Ptree = Build (SpecName) ;

```
Ptree.Name = SpecName ;
for i = 2 to Length(Pbase.Items) do    | Add the linguistic information.
{
        Rec = Pbase.Items[i];
        Obj = Ptree.item [Type =Rec.Type and Name = Rec.Name] ;
        if Obj is an operation or an action then
                Obj.[Description,Man]= Rec.[U_Def,Man]
        else
        Obj.[U_Name,Artc,U_Def,Plural,Man]= Rec.[U_Name,Artc,U_Def,Plural,Man]
}
Show (Ptree)
```

Data structures

```
SpecName: STRING ;
Pbase: PARABASE ;
i: INTEGER ;
Rec: CP[Type:INTEGER; Name,U_Name,Artc,U_Def,Man: STRING];
Obj:OBJECT ;
```

- Ptree = **Import_Paraphrase** (Ptree,BaseName,Btypes,Ctypes,Ops)

Precondition

```
FileExists(BaseName+'.Pdb') and ((Btypes=true) or (Ctypes=true) or (Ops=true));
```

Post Condition

```
Pbase = Open (BaseName) ;
if Btypes = true then
for every Rec: Rec ∈ Pbase.Data and (Rec.Type = BasicType) do
{
        Obj : Obj ∈ Ptree.Btypes and (Obj.Name =Rec.Name)
        Obj.[Artc,U_Name, Plural,U_Def,Man] = Rec. [Artc,U_Name,
Plural,U_Def,Man] ;
}
if Ctypes = true then
for every Rec: Rec ∈ Pbase.Data and (Rec.Type = ConstructedType) do
{
        Obj: Obj ∈ Ptree.Ctypes and (Obj.Name =Rec.Name)
        Obj.[Artc,U_Name, Plural,U_Def,Man] = Rec.
[Artc,U_Name,Plural,U_Def,Man] ;
}
if Ops = true then
for every Rec: Rec ∈ Pbase.Data and (Rec.Type = Operation ) do
{
        Obj: Obj ∈ Ptree.Ctypes and (Obj.Name =Rec.Name)
        Obj.Description = Rec.U_Def
}
```

- OutFiles = **Generate_Paraphrase** (Ptree,FileName,Format)

Precondition

Postcondition

| in Text format, the analyst has the choice not to save the generated paraphrase.

```
Empty (Paraph) ;
If (Format = Text) then
{
        Add (Paraph ,'Dictionary') ;
*       for all Obj: Obj∈ Ptree of class: data type or operation do
                Add(Paraph ,GlobalParaphrase (Obj)) ;
        Add(Paraph ,'Society Composition') ;
        for every agent Ag of the society do
        {
                Add(Paraph , GlobalParaphrase (Ag)) ;
                Add(Paraph ,'State components') ;
                if Ag.StateComps = nil then
                        Add(Paraph ,'This agent doesn't have any state component')
**              else for every state component Sc of Ag do
                        Add(Paraph , GlobalParaphrase(Sc)
                if Ag.Actions = nil then
                        Add(Paraph ,'This agent doesn't perform any action')
                else for every Action Ac of Ag do
                        Add(Paraph , GlobalParaphrase(Ac)
        }
        Display (Paraph) and Wait the decision of the analyst;
        if SaveOk then  OutFiles[1] = Save (Paraph, FileName+'.txt')}
Else            | Format HTML
{
        Empty (Paraph);
        Add to Paraph the needed header to generate an HTML file ;
        Do the steps in * ;
        for every agent Ag of the society do
        {
                if Ag is an agent then
                  s = 'Agent '+Ag.U_Name (as HTML link to Ag.Name+'.htm')
                else
                  s = 'Society '+Ag.U_Name (as HTML link to Ag.Name+'.htm')
                Add (Paraph,s) ;
        }
        OutFiles[1] = Save (Paraph, FileName+'.htm') ; i= 2 ;
        for every agent Ag of the society do
        {
                Empty (Paraph) ;
                Do the steps in ** ;
                OutFile[i] = Save (Paraph, Ag.Name+'.htm') ; i = i+1
        }
```

Data structures

Ag: AgentType
Paraph: SEQ[STRING]
s:STRING

- (Pbase,Dic) = **Save_Information** (Ptree,BaseName,Dic)

Precondition

Postcondition

```
If FileExists (BaseName+'.Pdb') then  Pbase = Open (BaseName) ;
Else
{ Pbase = Create ;Pbase.Name = BaseName}
Pbase.Items[1].Name = Ptree.Name;  | the path to the Albert II specification file.
For every object Obj in Ptree do
{
        if Obj is an action or an operation then
        {
                Rec.[Type,Name,U_Def,Man] = Obj.[Type,Name, Description,Man]
                Add(Pbase.Items,Rec);
        }
        else
        {
        Rec.[Name,U_Name,U_Def,Artc,Plural,Man] = Obj.[Name,U_Name,U_Def,
                                                        Artc,Plural,Man] ;
                Add(Pbase.Items,Rec);
                if (Obj.U_Def <>") and (Obj.U_Name ∉ U_Name(Dic.Data)) then
                        Add(Dic.Items,Obj.[U_Name,Plural,U_Def])}}
```

- lst = **Use_Dictionary** (DicName)

Precondition

Postcondition

| if the file *DicName* doesn't exist, this mean that we want to create a new dictionary.

```
If Not FileExists(DicName+'.dic') then
{
        Dic = Create (DicName) ; Empty (lst)          | new dictionary
}
Else
{
        Dic = Open (DicName);
        for i = 1 to Length(Dic.Items) do Add(Dic.Items[i].Name, lst)
}
```

- Obj = **Manipulate_Object** (Obj,Dic)

| We'll specify this functionality using the concept of Abstract Interaction Object AIO [Bodart97].

Precondition

```
Obj.Data<>nil
```

Postcondition

```
Spec = read only multi-line edit box (MBX) for the formal specification;
Spec.Text = Obj.Body;
AutoOk = check box (CHX) for 'yes' or 'non' to include the automatic paraphrase in the
output ;
AutoOk.Checked = Obj.AutoOk ;
ManOk = check box (CHX) for 'yes' or 'non' to include the manual paraphrase in the
output ;
ManOk.Checked = Obj.ManOk ;
Auto = read only multi-line edit box (MBX) for the automatic paraphrase;
Man = multi-line edit box (MBX) for the manual paraphrase;
Man.Text = Obj.Man;
If Obj is an operation or an action then
{
        Desc = extended edit box for the object description;
        Desc.Text = Obj.U_Def ;
        If Desc.Text<>" then Auto.Text = Paraphrase (Obj)
        else Auto.Text = " ;
}
Else
{
        Art = drop_down list box (DLB) with the items 'a', 'an' and 'the' for the article;
        Article.Text = Obj.Artc;
        Name = drop_down list box for the linguistic name ;
        Name.Items = Open (Dic);
        If Obj.U_Name <>" then Name.Text = Obj.U_Name
        Else    Name.Text = Obj.Name        | If there is no linguistic name, use the formal one.
        Plural = edit box  for the plural name;
        If Obj.Plural <>" then Plural.Text = Obj.Plural
        Else    Plural.Text = Name.Text + 's' (or es or ies)    | some text analysis.
        Denotation = edit box  for the denotation;
        Denotation = Obj.U_Def ;
        Auto.Text = Paraphrase (Obj)

        | get linguistic information from the current dictionary.

        If Name.Text is modified and (Name.Text ∈ Name.Items) then
        {
                Rec = Dic.Items[Name = Name.Text];
                Plural.Text = Rec.Plural ; Denotation.Text = Rec.Denotation;
        }}
```

- **Show** (Ptree)

| to specify the way of displaying the parse tree, we use the AIO OUT. it has many attributes and primitives, but
| we'll only indicate those that we use.

```
Outline control = CP [Items : SET [CP[Text : STRING ; Data : POINTER]] ; Att2,....]
```

Precondition

Postcondition

```
Clear (Outt.Items) ;
It.Text = 'BASIC TYPES'; It.Data = nil;
Outt.AddChild (It,Outt)                       | Add It at the top level of Outt.
Pt = Ptree.BTypes ;
while Pt<>nil do                              | Insertion of basic data types.
{
        It1.Text = Pt^.Name; It1.Data = Pt;
        Outt.AddChild (It1, It)               | Add It1 inside the Basic Types.
}
It.Name = 'CONSTRUCTED TYPES'; It.Data = nil; Outt.AddChild (It, Outt) ;
Pt = Ptree.CTypes ;
while Pt<>nil do                              | Insertion of constructed data types
{
        It1.Text = Pt^.Name; It1.Data = Pt; Outt.AddChild (It1, It) ;
        Pt1 = Pt^.Intype ;                    | get the interior data type.
```

| To minimize the effort of the analyst, we try to find if the interior data type was already inserted in the tree to
| avoid the insertion of the same object at many levels. If it was, we need just to direct the actual pointer towards
| it. We do this only when the interior data type is a basic one (like NAMES = SET [NAME]).

```
        If Pt1 is a basic type then
        {
                Pc1 =  Search (Pt1^.Name,Outt);
                if not found then                     | New interior type, so we insert it.
                {
**                      It2.Text = Pt1^.Name; It2.Data = Pt1;
                        Outt.AddChild (It2, It1) ;
                }
                Else  Pt^.Intype = Pc1 ;              | Found, so reuse it.
        }
        Else
        {
                Pt1^.Name = Pt^.Name+'_'+Pt1^.Typ;     | fictive name.
                Insert the type Pt1 ;
        }}
```

| The insertion of constructed data types may continue recursively if the interior type is also a constructed one, but
| here, we'll stop at the second level hopping that we have made our point.

```
It.Text = 'OPERATIONS';
It.Data = nil;
Outt.AddChild(It,Outt);
Pt = Ptree.Operations ;
while Pt <>nil do         | Insertion of operations
{
        It1.Text = Pt^.Name; It1.Data = Pt;
        Outt.AddChild (It1, It) ;
        Pt1 = Pt^.Domain ;
        while Pt1 <>nil do the steps in ** ;
        Pt1 = Pt^.Codomain ; Do the steps in ** ;
```

```
}
Pt = Ptree.Societies ;
It.Text = 'Society'+Pt^.Name ;          | name of the main society.
It.Data = Pt;
Outt.AddChild(It,Outt);                 | at top level.
while Pt<>nil do
{
Pt1 =Pt^.Agents ;
while Pt1<>nil do
{
        It1.Text = 'Agent'+Pt1^.Name ;       It1.Data = Pt1;
        Outt.AddChild(It1,It);
        Pt2 =Pt1^.StateComps ;
        if Pt2<>nil then
        {
                It2.Text = 'State components' ;It2.Data = nil;
                Outt.AddChild(It2,It1);
                while Pt2<>nil do
                {
                        It3.Text = Pt2^.Name ; It3.Data = Pt2;
                        Outt.AddChild(It3,It2);
                }
        }
        Pt2 =Pt1^.Actions ;
        if Pt2<>nil then
        {
                It2.Text = 'Actions' ;It2.Data = nil;
                Outt.AddChild(It2,It1);
                while Pt2<>nil do
                { It3.Text = Pt2^.Name ; It3.Data = Pt2; Outt.AddChild(It3,It2);}}}
```

Data structures

```
Pt,Pt1,Pt2,P: POINTER;
It,It1,It2,It3: CP [Text: STRING; Data: POINTER]
```

- s = **GlobalParaphrase** (Obj)

Precondition

| The analyst has the possibility to exclude an object if he doesn't enable any of its paraphrase forms.

```
Obj.AutoOk or Obj.ManOk
```

Postcondition

```
s = '' ;
If Obj.AutoOk then   s = Paraphrase (Obj) ;
If Obj.ManOk then    s = s+ Obj.Man
```

Data structures

```
s : STRING
Obj : OBJECT
```

- Output = **Paraphrase** (Obj)

The automatically generated paraphrase of an object is the result of applying the template-based techniques on the content of the associated template. For each class of objects we define the procedure to be applied on the template of an object of this class.

For example, to generate an analyst-oriented paraphrase for the enumerated data type object, we apply the following algorithm:

Start with Output = Formal Name
If there is a linguistic name then Add ' (Linguistic Name)'
If there is no linguistic denotation then Add ' is defined as one of the following constant values:'
Else Add linguistic denotation then Add ' is defined as one of the following constant values:'
For every value of the object Add its name followed by ',' to Output
Cut the ',' at the end of Output
Replace the ',' by the word ' or ' between the two last values {some text manipulation}

Lets apply this algorithm on the object (Sts = Enum [raw, processed]). We suppose that the analyst provides the linguistic information:

name= Status.
Denotation= status of the piston.

Ouput='Sts'
Output='Sts (Status): '
Output='Sts (Status): status of the piston. It is defined as one of the following constant values:'
Output='Sts (Status): status of the piston. It is defined as one of the following constant values: raw,'
Output='Sts (Status): status of the piston. It is defined as one of the following constant values: raw, processed,'
Output='Sts (Status): status of the piston. It is defined as one of the following constant values: raw, processed'.
Output='Sts (Status): status of the piston. It is defined as one of the following constant values: raw or processed'.

For some object classes we may apply the paraphrasing procedure recursively to obtain the complete paraphrase of an object. This is the case of most of constructed data types.

5.3.2 Type PARSETREE

The parse tree holds the information found during the analysis of the treated Albert II specification. So, we defined it as a set of linked lists. Each list contains the founded objects of one class of Albert II Formal objects.

Interface

Build: STRING → PARSETREE

PARSETREE = CP[Name: String ;
Btypes:SET[BTypeObject] ;
Ctypes : SET[TypeObject] ;
Operations : SET [OperationObject] ;

```
                          Societies : SET[SocietyObject]]

CTypeObject  = CP[  F_Name, U_Name, Plural, U_Def : STRING;

                    Auto,Man: STRING ;
                    Artc: [a, an, the] ;
                    Type: INTEGER ;
                    Value: SET[TypeObject] ;
                    AutoOk,ManOk:BOOLEAN
                    ]

BTypeObject  = CP[  Name, U_Name, Plural, U_Def,Auto,Man : STRING;
                    Artc: [a, an, the] ;
                    AutoOk,ManOk:BOOLEAN]

OperationObject = CP[Name,Description, Auto, Man: STRING ;
                    Domain : SET [UNION [BTypeObject, CTypeObject]] ;
                    Codomain: UNION [BTypeObject, CTypeObject] ;
                    AutoOk,ManOk:BOOLEAN]

SocietyObject = CP [  Name, U_Name, Plural, U_Def, Auto, Man: STRING ;
                    Artc: [a, an, the] ;
                    Composition: SET [STRING] ;
                    Agents: SET [AgentObject] ;
                    AutoOk,ManOk:BOOLEAN]

AgentObject = CP [  Name, U_Name, Plural, U_Def, Auto, Man: STRING ;
                    Artc: [a, an, the] ;
                    StateComps: SET [CompObject] ;
                    Actions: SET [ActionObject] ;
                    AutoOk,ManOk:BOOLEAN]

CompObject = CP[    Name, U_Name, U_Def, Auto, Man: STRING ;
                    Artc: [a, an, the] ;
                    OfType1,OfType2: STRING ;
                    DeriveFrom, ExportTo: SET[STRING] ;
                    AutoOk,ManOk:BOOLEAN]

ActionObject = CP [  Name,Description: STRING
                    Arguments, ExportTo: SET[STRING]]
```

We can see that there are a lot of common information in the structures of these objects, and we can use this fact later in the physical architecture.

Specification

Ptree = **Build** (SpecName)

Precondition

The specification is syntactically correct.

Postcondition

Build a parser using the adopted tool;
Add to the parser the needed actions to construct the parse tree;
Ptree = Parse (SpecName)

*5.3.3 Type **DICTIONARY***

Interface

Open: DICTIONARY → DICTIONARY
Create: → DICTIONARY

DICTIONARY = CP[Name:STRING ; Items : SET[CP[Name, Plural, Denotation: STRING]]]

Specification

Dic = **Open** (Dic1)

Precondition

Postcondition

Dic.Name = Dic1.Name ;
Dic.Items = Dic1.Items ;

Dic = **Create**

Precondition

Postcondition

Dic.Name = '';
Dic.Items = [] ;

*5.3.4 Type **PARABASE***

Interface

Open: PARABASE → PARABASE
Create: → PARABASE

PARAPHRASE = CP[Name: STRING ;
Items : SET[CP[Type,Name,U_Name,Plural,U_Def: STRING ;
Artc: STRING[3]]]]

Specification

Pbase = **Open** (Base)

Precondition

Postcondition

Pbase.Items = Base.Items;
Pbase.Name = Base.Name ;

Pbase = **Create**

Precondition

Postcondition

Pbase.Name = '' ;
Pbase.Items = []

Remarks on the logical architecture

- [Dubois97] presents the logical architecture as divided into six levels: coordination units(6), Human-Machine Interface units (5), treatments units (4), persistent data units (3), system/tool units (2) and OS units (1). In our architecture we didn't respect this architecture. In fact, the coordinator unit implicitly includes the HMI units.
- Why we keep the path to the specification's file when we save its paraphrase? In fact we have two choices: save the structure of the parse tree with the linguistic information, or save a path to the file, and when we consult the paraphrase, we rebuild the parse tree from the specification. We preferred the second choice because we estimated that rebuilding the tree is more practical than saving its structure which needs to define complex data structures and procedures to save and to read this structure.
- Why we don't import the actions. In fact, we estimated that, when we write a new specification, it is more probable to redefine the same data types with identical same structures, and the same operations on these types, than to redefine the same actions with the same names and parameters. {not very convincing!}

5.4 The Physical Architecture

After selecting the programming environment that will be used to implement the paraphrazer, the logical architecture is concretized by the physical architecture. We use the Borland Delphi environment (Supporting Pascal programming language) to develop the interface of our paraphrazer (so we will use Concrete Interaction Objects (CIO)), and we use Visual Parse++ to define the lexical and syntax analyzers. Visual Parse++ generates a Delphi-like unit containing the parser used by the paraphraser to analyze the treated Albert II specification.

5.4.1 Unit COORDINATOR

The functions of the coordinator unit are provided to the analyst through the interface that can be easily developed using the predefined Delphi objects. We use the object **TMainMenu** to define our menu of figure 5.3.

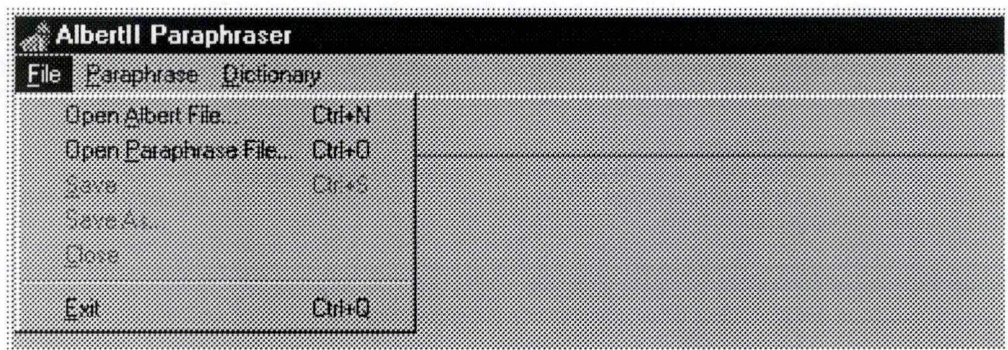

Figure 5.3: Paraphrazer main menu

The source code of this unit in annex (Main.pas) contains the implementation of the functionalities specified above in the logical architecture.

- **Functionality:** Paraphrase_New_Specification (TMainWin.New1Click)

Enable the analyst to provide the specification file name via the FileOpen dialog box ;
(the file filter is limited to *.txt files).
if OpenOk then
{
Parse the specification and build the parse tree (AnalyseSpec);
Display the parse tree (EmptySpec) (Figure 5.4);
Activate the related functionalities (File.save, File.save as, File.close, Paraphrase.Generate,
Paraphrase.Import, Dictionary.Use)
}

- **Functionality:** Open_Old_Paraphrase (TMainWin.OpenParaphrase1Click)

Enable the analyst to provide the paraphrase file name via the FileOpen dialog box ;
(the file filter is limited to *.Pdb files).
if OpenOk then
{
Open the file ;
Get the path to the original specification file from the first record of the opened data base.
AnalyseSpec;
EmptySpec;
Add the linguistic information in the data base to the parse tree (ReadInfo);
Enable the related functionalities
Open the default dictionary (UseDictionary)}

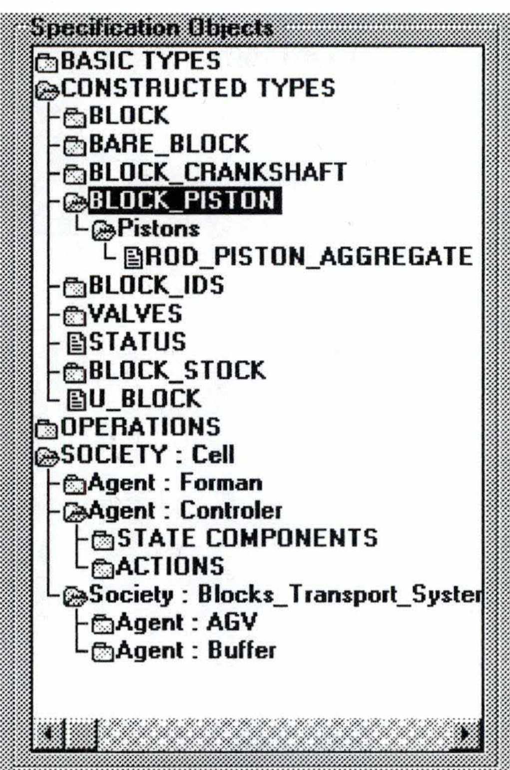

Figure 5.4: Displaying the content of the parse tree

- **Functionality:** Manipulate_object (TMainWin.ArbClick)

  The modification of linguistic information becomes possible as we display the parse tree.

  Select an object Obj1;
  Display the related information in the corresponding CIOs;
  If Select another object Obj2 then Save contents of the CIOs to Obj1 ;

  The analyst can modify the information via the window shown in figure 5.5.

- **Functionality:** Use_Dictionary (UseDictionary)

  Enable the analyst to provide the dictionary name via the FileOpen dialog box ;
  (the file filter is limited to *.dic files).
  Add the names of the dictionary items to the CIO used to provide the object name (see Figure 5.5).

- **Functionality:** Save_Information (TMainWin.Save1Click)

  If the specification is paraphrased for the first time then
  {
  Enable the analyst to provide the file name via the FileOpen dialog box ;
  (the file filter is limited to *.Pdb files) ;
  Save the linguistic information (SaveSpec);
  If we have new definitions then add them to the current dictionary}

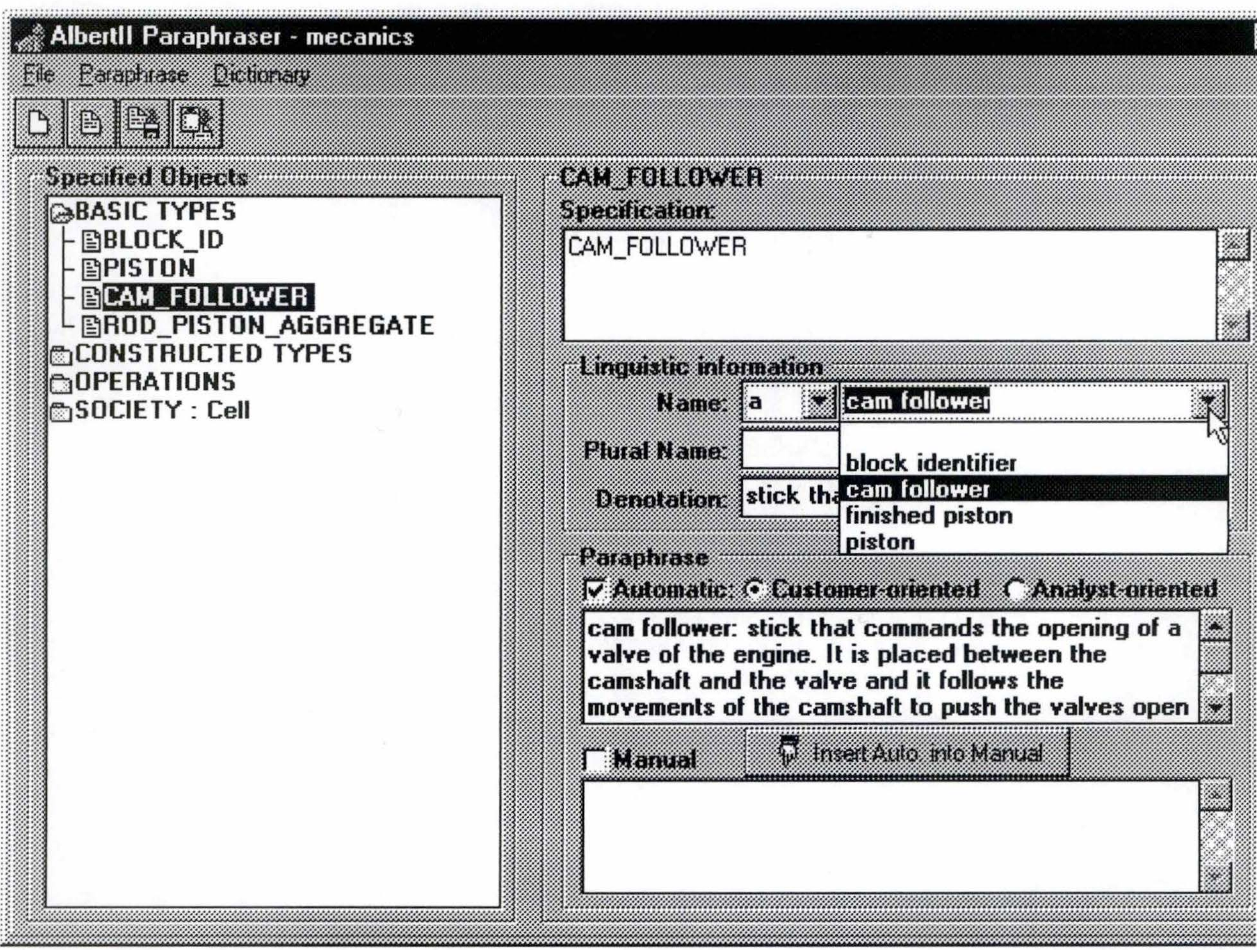

Figure 5.5: Paraphraser Main Window

- **Functionality:** Import_Paraphrase (TMainWin.Import1Click)

  Enable the analyst to provide the dictionary name via the dialog box shown in figure 5.6 ; Import the linguistic information of the sdelected classes.

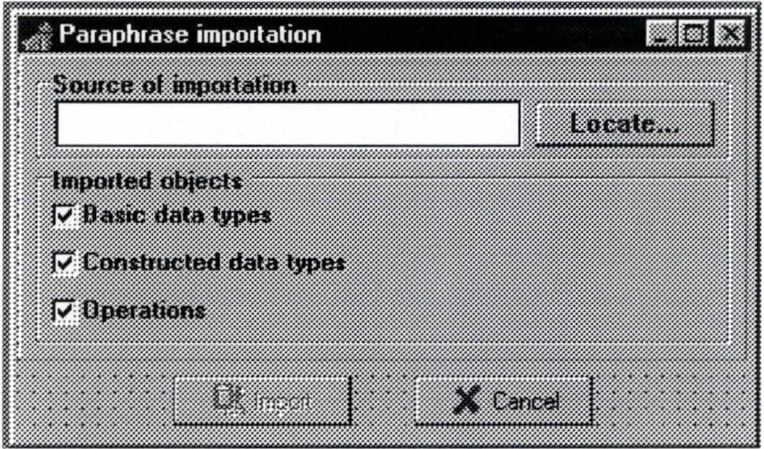

Figure 5.6: Paraphrase importation dialog box

- **Functionality:** Generate_Paraphrase (TMainWin.GenerateText1Click)

We provide the dialog box of figure 5.7 to enable the analyst to select the format of the output and the composition of the generated paraphrase.

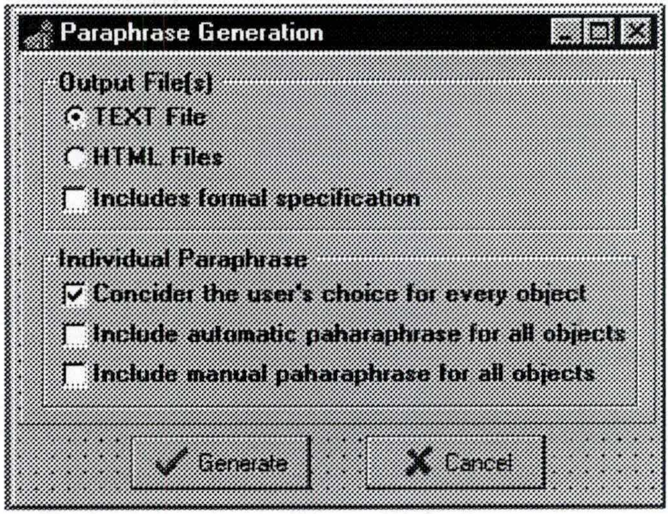

Figure 5.7: Paraphrase generation dialog box

5.4.2 Parse Tree Unit

```
Spec = record
        BTypes, CTypes,Operations,Societies:TList;
     end;
```

The CIO **TList** defines a list of objects (any kind of object). In our case, we defined an object grouping all the kinds of objects specified in 5.3.2.

```
MyObject=record
        Name,U_Name,Plu,U_Def,Body,Man:string;
        TheAn:string[3];
        IsPlu,Star,Undir,Manual,Auto:Boolean;
        case typ:integer of
           1,15:(Rd:String);
           2,3,4,5,6,7:(Value:TList);          {Types:Seq,Set,Enum,Union,Cp,Bag}
           8,9:(Dom,Codom:TList);                     {Operations and Table types }
           10:(SocAgents,Agents:TList);               {Societies}
           11:(StateComps,Actions:TList);   {Agents}
           12:(Fix:Boolean;CompType,DerCnt:Integer;OfType1,OfType2:String;
              Deriv,Exprt:TStringList);        {State Components}
           14:(Args,AExprt:TStringList);              {Actions}
        end;
```

We see that in some cases, there are some unused fields (like U_Name for operation), but we were forced to do defined a general object because Visual Parse++. In fact, in Visual Parse++, we must define the so called stack-value, which is pushed in or popped out the objects stack during the parsing process. We can used only one type for this value, so we must define it in a way to be able to keep the information about the deferent kinds of objects encountered during the parsing process.

Source code of this unit (Generate.pas) is in Annex C.

- **Functionality:** Build (AnalyseSpec)

This functionality is realized by defining the lexical and syntax analyzers, then generating the parsing function by Visual Parse++, then adding the actions to build the parse tree (in the function AYaccClass.Reduce). These procedures are described next (section 5.5).

5.4.3 The dictionary/ The parabase types

We use the CIO TTable to perform the creation and the manipulation of the dictionary and linguistic information base (Unit Main.pas).

5.5 The Parser Implementation

5.5.1 Tokens of the lexical analyzer

The lexical analyzer is built by analyzing the abstract syntax of the Albert II language to define the tokens of our sub-language (we treat only a part of the language). This syntax is found in the Albert II reference manual [Du97].

In annex A we find the description of the tokens that our lexical analyzer must recognize. Here, we are describe the reasons of choosing these tokens.

When we presented the Albert II language (Chapter two), we saw that an Albert II specification is an ensemble of sections that each of which has a header and an end marker. For example %BASIC TYPES and | END_BASIC_TYPES for the section of basic types. The end markers must be recognized as strings and then ignored because they are needed just to tell where every section ends. Visual Parse++ enables use to define the regular expressions describing these tokens and to precise that the must be recognized but ignored. So, in the %expression of our Visual Parse++ rule file, we have the following expressions:

```
'\| END_BASIC_TYPES\n'              %ignore;
'\| END_CONSTRUCTED_TYPES\n'        %ignore;
'\| END_OPERATIONS\n'               %ignore;
'\| END_SOCIETY\n'                  %ignore;
'\| END_STATE_COMPONENTS\n'         %ignore;
'\| END_ACTIONS\n'                  %ignore;
'\| BASIC_CONSTRAINTS\n'            %ignore;
'\| END_AGENT\n'                            %ignore;
'\| END_CONSTRAINTS\n'              %ignore;
'\| END_SPEC\n'                     %ignore;
```

The sections headers are needed as tokens because some sections may be empty, so the corresponding production in the grammar will have just the section header in it right side. So the headers are described by regular expressions that are assigned names to be used in the grammar. We have the following regular expressions:

```
'%SPEC'                     SpecStart;
'%BASIC TYPES'              BasicTypes;
'%CONSTRUCTED TYPES'        ConsTypes;
'%OPERATIONS'               Ops;
'%SOCIETY'                  Society;
```

```
'%AGENT'                          Agent;
'%STATE COMPONENTS'               StateComp;
'%ACTIONS'                        Actions;
```

As we ignore a part of a specification, we must ignore the sections of this part which starts at the %DERIVED COMPONENTS section. In visual Parse++ this gives the following expressions:

```
'%DERIVED COMPONENTS'             %ignore, %push IgnoredPart;

%expression IgnoredPart
'.'                               %ignore;
'\n'                              %ignore;
'\| END_AGENT'                    %ignore, %pop;
```

Finally, we have the tokens of all the keywords, operators, special symbols, etc. (See annex A).

5.5.2 Description of the grammar

after the realization of the lexical analyzer, we must defined our syntax analyzer by defining a BNF grammar of the treated part of an Albert II specification. Visual Parse++ enabled us to test this grammar and to organize its productions to obtain a grammar without any conflict.

Next, we are going to present of this grammar.

- An Albert II specification starts by the name of the specified society, then the specification of the data types and operations, and then the specification of the agent of the society.

```
%production spec

spec          -> SpecStart Identifier TypesOpsDec RootSoc;
```

- A basic type is an identifier. Our basic types list contains zero or more basic type.

```
BasicDec      -> BasicTypes;              {the header of basic types section}
BasicDec      -> BasicTypes BTypeList;
BTypeList     -> BasicType;
BTypeList     -> BasicType BTypeList;
BasicType     -> Identifier;
```

- Our constructed types list contains zero or more type.

```
ConstDec      -> ConsTypes;
ConstDec      -> ConsTypes CTypeList;
CTypeList     -> ConstType;
CTypeList     -> ConstType CTypeList;
ConstType     -> Identifier Equal TypeExpr;
```

- We have seven constructed data types: Redefined, Cartesian product, Set, Sequence, Bag, Table, Union and Enumerated data types.

```
TypeExpr      -> Identifier;
TypeExpr      -> CP LeftBracket Selector RightBracket;
TypeExpr      -> Set LeftBracket TypeExpr RightBracket;
TypeExpr      -> Seq LeftBracket TypeExpr RightBracket;
TypeExpr      -> Bag LeftBracket TypeExpr RightBracket;
TypeExpr      -> Table LeftBracket TypeExpr RightArrow TypeExpr RightBracket;
TypeExpr      -> Union LeftBracket TypeList RightBracket;
TypeExpr      -> Enum LeftBracket TypeList RightBracket;
```

- A Cartesian product data type is a list of fields defined as a constructed data types (but Name ':' body instead of Name '=' body).

```
Selector      -> Identifier Column TypeExpr;
Selector      -> Identifier Column TypeExpr Comma Selector;
```

- We paraphrase the operations on data types. In Albert II language, the definition of an operation consists of two parts: the header of the operation and a possible definition of the a particular effects of this operation on its arguments. In our paraphraser, we ignore the second part.

```
OpDec         -> Identifier Column OpArity;
OpDec         -> Comment Identifier Column OpArity;
OpArity       -> RightArrow TypeExpr;
OpArity       -> OpDom RightArrow TypeExpr;
OpDom         -> TypeExpr;
OpDom         -> TypeExpr CartProd OpDom;
```

- Society specification: first its composition. A society may be composed of agents and sub-societies.

```
RootSoc       -> SocList;
SocList       -> SocDec AgentList;
SocList       -> SocDec AgentList SocList;
SocDec        -> Society SocAgentName SubSocAgList;
SubSocAgList  -> SubSocAg;
SubSocAgList  -> SubSocAg SubSocAgList;
SubSocAg      -> LeftParen Identifier RightParen; {one agent or sub-society}
SubSocAg      -> LeftParen Identifier RightParen RightParen RightParen; {several...}
```

- As we said, we paraphrase only the state components and the actions of an agent.

```
AgentList     -> AgentDec;
AgentList     -> AgentDec AgentList;
AgentDec      -> Agent SocAgentName StateCompDec ActionsDec;
```

- The name of and agent may be an identifier (the name of the main society) or what we call a path name. For example Cell.Controler is the name of the agent Controler which is a member of the society Cell.

```
SocAgentName -> Identifier;
SocAgentName -> Identifier Dot SocAgentName;
```

- State Components list

```
StateCompDec        -> StateComp;
StateCompDec        -> StateComp CompList;
CompList            -> CompDec;
CompList            -> CompDec CompList;
```

- A state component that has constant value is specified by preceding it name by *.

```
CompDec     -> Star Comp Derivs Exports;
CompDec     -> Comp Derivs Exports;
```

- A state component may be derived from other state components.

```
Derivs      -> ;
Derivs      -> Derived CompNameList;
```

- The name of a state component is an identifier (first production of the declaration of SocAgentName).

```
CompNameList-> SocAgentName;
CompNameList -> SocAgentName Comma CompNameList;
```

- A state component may be exported to other agents of the society.

```
Exports     -> ;
Exports     -> RightArrow CompNameList;
```

- We have four types of state components: Set, Sequence, Instance and Table.

```
Comp        -> Identifier CompType Identifier;
Comp        -> Identifier TableComp Identifier Index Identifier;
CompType    -> SetComp;
CompType    -> SeqComp;
CompType    -> InstanceComp;
```

- Actions

```
ActionsDec  -> Actions ;
ActionsDec  -> Actions ActionList;
ActionList  -> ActionDec;
ActionList  -> ActionDec ActionList;
```

- An action may be dependent of other actions. It must be executed always in combination with them.

```
ActionDec   -> Star Identifier ActionBody Exports;
ActionDec   -> Identifier ActionBody Exports;
```

- Actions can be without or with arguments.

```
ActionBody  -> ;
ActionBody  -> '(' TypeList ')';
```

5.5.3 The parser

After defining the lexical and the syntax analyzers, Visual Parse++ enables us to generate automatically a file (FileName.ypa) containing all the needed information about the parser. In particular, it contains a procedure (Reduce) that has the structure of the parse tree. In our case, the generated file is Delphi-oriented because we implement our paraphraser using Borland Delphi programming environment.

To complete the realization of our parser, we must add the following information to this file:

- *Stack value structure*

Visual Parse++ uses a stack structure during the Shift-Reduce parsing. At any moment, we can push in or pop out the value on the top of the stack. Visual Parse++ defines a default type for this value. It is

```
AYaccStackElement = class(SSYaccStackElement)
 public
 Value  : MyObject;

 constructor Create;
end;
```

We added the field Value with our own type to be able to put in it the information that we need to have during the parsing process. The type MyObject is the definition of the structure that we use to keep information about the detected objects in the treated specification (5.2.1)

- *Parsing function*

```
function Analyze(FileName:PChar): String;
var
 Lexer       : SSLex;                  {Lexical analyzer}
 Parser      : AYaccClass;             {Syntax analyzer}
 LexTable    : SSLexTable;             {Tokens table}
 ParseTable : SSYaccTable;             {Productions table}
 Consumer    : SSLexFileConsumer;      {Input string}

begin

{FileName is the name of the input file containing the specification}

 Consumer := SSLexFileConsumer.Create(FileName,32768,0,SSLexTextMode);

{albert.dfa and albert.llr are generated automatically by Parse++}

 LexTable := SSLexTable.Create( 'c:\memoire\albert.dfa');
 Lexer := SSLex.Create( Consumer, LexTable);
 ParseTable := SSYaccTable.Create( 'c:\memoire\albert.llr');
 Parser := AYaccClass.CreateLex( Lexer, ParseTable);
 Parser.Parse;                                          {Main parsing call}
 Result := Parser.FinalValue.name;
```

end;

For more details about the signification and use of predefined functions, see the reference manual of Visual Parse++}

- *Parsing actions*

Visual Parse++ generates the Reduce function that the parser calls every time it recognized and build a substring that matches a production in the grammar. During the parsing process, we must construct our explicit parsing tree representation. We do this by adding the actions that the parser must perform when it reduces every production. For example, when it recognizes a basic type, it must store it in the list of basic types. In the Reduce function, we write the actions for this task when reducing the corresponding production (See annex B). For example, the performed actions after the recognition of a Fixed state component are as follow:

```
AYaccFixedCompNo:
  { CompDec -> Star Comp Derivs Exports }
  begin
    E:= AYaccStackElement(StackElement); {Create an empty stack element that will be returned as a result}
    E1:= AYaccStackElement(ElementFromProduction(1)); {Get the information about the component}

    E2:= AYaccStackElement(ElementFromProduction(2)); {Get the derivation list if it is a derived component}
    E3:= AYaccStackElement(ElementFromProduction(3)); {Get exportation list if exported to other agents}
    New(PE);
    E.Value:=E1.Value;
    E.Value.Fix:=True;
    E.Value.Deriv:=TStringList.Create;
    E.Value.Exprt:=TStringList.Create;
    E.Value.Deriv:=E2.Value.Deriv;
    E.Value.Exprt:=E3.Value.Exprt;
    E.Value.Body:='*'+E1.Value.Body+E2.Value.Body+E3.Value.Body;{Get the specification of the object}
    Result:=E
  end
```

We write the actions that the parser must perform at every reduction. An important action that is repeated very much is the access to the deferent components of a production:

AYaccStackElement(ElementFromProduction(n)) with n >= 0

At the end of the parsing process, we are supposed to have all the information found in the input specification. These information will be stored in a structure that we defined for this purpose. It is the explicit representation of the parse tree of the specification.

To include the parser in our paraphrase, we need just to call the function *Analyze* by passing the name of the input file. If the parsing terminates without errors, we will find our parse tree in a global variable of type *Spec*.

5.5.4 Realization

The realization of the paraphraser took about three months. The main units are joint in annex C.

Conclusion

Conclusion

A) Paraphraser evaluation

In this work, we proved that paraphrasing an AlbertII specification is possible. In fact, although the fact that we didn't handle all the Albert II language objects, but our work shows that we can generate a paraphrase of good quality with a minimum of extra-information, because the specification provides us with a lot of information.

It was clear that The proposed information (which information and under which form) to be provided by the analyst, has a very important role in determining the quality of the paraphrasing result.

It was also clear that the quality of the generated paraphrase depends on the paraphrasing patterns of each class of objects (so on the person who proposes them). In fact, as in reality, it is difficult to be completely satisfied (and more difficult to satisfy the others) of the generated paraphrase, but we find this normal because we can say the same thing in many ways.

In annex we have the paraphrase of our very simple case study. Unfortunately, only about 85% of this paraphrase was automatically generated, and it needed our intervention to complete it.

B) Futur works

We think that there are many interesting ideas to be realized in future works:

- *paraphrasing a complete Albert II specification:* in fact, the Albert II language specifies a system in a way that the information about an object can be distributed over many sections. For example, specifying an action is done it the %actions section, the %Triggerings section, the %effects of action section, etc. It would be very interesting to put all such information in one place.

- *Generating multi-language paraphrase*: we hope that one day, the Albert II language will be adopted in many societies that may be of different cultures (English, French, etc.). It would be interesting to enable the generation of a multi-language paraphrase of the same English specification.

- *Domain paraphrasing*: in our paraphraser we use a dictionary containing definitions of some elements that was used in paraphrasing old specifications, and we try to reuse these elements to minimize the effort of the analyst. We can imagine a more sophisticated approach to exploit its acquired knowledge.

References

References

[Aho79]
Alfred V.Aho, Heffrey D.Ullman, "Principles of computer design ", *ADDISON-WESLEY PUBLISHING COMPANY, New Jersey, USA,* April 1979.

[Bodart97]
F.Bodart, "Interface Homm-Machine", Course notes for the second year, *Institut d'Informatique/FUNDP*, Academic year 1996-1997.

[Dalia95]
Hercules Dalianis, "Aggregation, Formal Specification and Natural Language Generation", *In Proceedings of the NLDB'95, First International Workshop on the Applications of Natural Language to Data Bases, pp 135-149, Versailles, France,* June 28-29, 1995.

[Du97]
Philippe Du Bois, "The Albert II Reference Manual Language Constructs and informal Semantics, Version 2.0", *The institute of computer science, University of Namur, Belgium,* April, 1997.

[Dubois97]
E.Dubois, "Méthodologie de développement de logiciels", Course notes for the second year, *Institut d'Informatique/FUNDP*, Academic year 1996-1997.

[Elhadad93]
Michael Elhadad, "Using Argumentation to control Lexical Choices : A Functional Unification Implementation", *The Graduate School of Arts and Sciences, Columbia University, USA*, 1993.

[Kraynak97]
"Internet 6-in-1",*QUE, A Division of Macmillan Computer Publishing, 201 West 103rd Street, Indianapolis, Indiana 46290 USA.*

[Reiter95]
Ehud Reiter, "NLG vs. Templates", *CoGenTex, Inc, USA,* 1995.

[Reiter96]
Ehud Reiter, "Building Natural Language Generation Systems", *Department of Computer science, University of Aberdeen,King's College, Aberdeen, BRITAIN,* 1996.

[Robin98]
"Topics in Artificial Intelligence: Natural Language Generation",
http://www.di.ufpe.br/~jr/teaching/nlg/syllabus.html

[Rolla92]
Collette Rolland & Christophe Proix, "A Natural Language approach for Requirements Engineering", *CAISE-92 Int. Conf. on Advanced Information Systems Engineering, (Ed.) P. Loucopoulos, Springer Verlag Lecture Notes in Computer Science, no 593, pp. 257-277,* 1992.

[Sand95]
"Visual Parse++ Version 2.0 Guide and Reference", SandStone Technology Inc, 1994-1995.

ANNEXES

Annex A

The case study

(Specification and Constumer-oriented paraphrase)

```
%SPEC Cell
  %BASIC TYPES
      BLOCK_ID
      PISTON
      RING
      ROD
      PEG
      CAM_FOLLOWER
  | END_BASIC_TYPES
  %CONSTRUCTED TYPES
      BLOCK=UNION[BLOCK_CRANKSHFT,BLOCK_VALVE,FINISHED_BLOCK]
      BARE_BLOCK=CP[Id:BLOCK_ID,BearingsOn:BOOLEAN,Checked:ENUM[not,
      good,bad]]
      BLOCK_CRANKSHAFT=CP[Block:BLOCK_PISTON,Crank:SET[CRANKSHAFT],
      ScrewedOk: BOOLEAN] BLOCK_PISTON=CP[Block:BARE_BLOCK,Pistons:
      SET[ROD_PISTON_AGGREGATE]]
      BLOCK_IDS=SEQ[BLOCK_ID]
      STATUS=ENUM[raw,processed]
      BLOCK_STOCK=table[BLOCK_ID ->BLOCK]
  | END_CONSTRUCTED_TYPES
  %OPERATIONS
    Removal: CAM_FOLLOWERS_PALLET \x SET[CAM_FOLLOWER] \->
CAM_FOLLOWERS_PALLET
  | END_OPERATIONS
  %SOCIETY Cell
    (Foreman)
    (Controller)
    (Blocks_Transport_System)
  | END_SOCIETY
    %AGENT Cell.Foreman
      %ACTIONS
      DetermineDailyPlan(BLOCK_IDS) -> Cell.Controller
      | END_ACTIONS
      | BASIC_CONSTRAINTS
      %DERIVED COMPONENTS
      %INITIAL VALUATION
      | DECLARATIVE CONSTRAINTS
      %STATE BEHAVIOUR
      %ACTION COMPOSITION
      %ACTION DURATION
      | OPERATIONAL CONSTRAINTS
      %PRECONDITIONS
      %EFFECTS OF ACTIONS
      %TRIGGERINGS
      | COOPERATION CONSTRAINTS
      %ACTION PERCEPTION
      %STATE PERCEPTION
      %ACTION INFORMATION
      %STATE INFORMATION
      | END_CONSTRAINTS
    | END_AGENT
    %AGENT Cell.Controller
      %STATE COMPONENTS
```

```
    Finished \sequence-of BLOCK_ID  -> Cell.Foreman
    Semi_finished \sequence-of BLOCK_ID  -> Cell.Foreman
    | END_STATE_COMPONENTS
    %ACTIONS
    *RequestTransport(BLOCK) -> Cell.Blocks_Transport_System.AGV
    | END_ACTIONS
    | BASIC_CONSTRAINTS
    %DERIVED COMPONENTS
    %INITIAL VALUATION
    | DECLARATIVE CONSTRAINTS
    %STATE BEHAVIOUR
    %ACTION COMPOSITION
    %ACTION DURATION
    | OPERATIONAL CONSTRAINTS
    %PRECONDITIONS
    %EFFECTS OF ACTIONS
    %TRIGGERINGS
    | COOPERATION CONSTRAINTS
    %ACTION PERCEPTION
    %STATE PERCEPTION
    %ACTION INFORMATION
    %STATE INFORMATION
    | END_CONSTRAINTS
  | END_AGENT
  %SOCIETY Cell.Blocks_Transport_System
    (AGV)))
    (Buffer)
  | END_SOCIETY
    %AGENT Cell.Blocks_Transport_System.AGV
      %STATE COMPONENTS
      Transported \instance-of BLOCK  -> Cell.Foreman
      | END_STATE_COMPONENTS
      %ACTIONS
      *Load(BLOCK)
      *Transport(BLOCK)
      Transportation
      | END_ACTIONS
      | BASIC_CONSTRAINTS
      %DERIVED COMPONENTS
      %INITIAL VALUATION
      | DECLARATIVE CONSTRAINTS
      %STATE BEHAVIOUR
      %ACTION COMPOSITION
      Transportation <-> c.RequestTransport(_),Load(_),Transport(_)
      %ACTION DURATION
      | OPERATIONAL CONSTRAINTS
      %PRECONDITIONS
      %EFFECTS OF ACTIONS
      %TRIGGERINGS
      | COOPERATION CONSTRAINTS
      %ACTION PERCEPTION
      %STATE PERCEPTION
      %ACTION INFORMATION
```

```
        %STATE INFORMATION
        | END_CONSTRAINTS
      | END_AGENT
      %AGENT Cell.Blocks_Transport_System.Buffer
        %STATE COMPONENTS
        Content \set-of BLOCK  -> Cell.Controller
        *Capacity \instance-of INTEGER  -> Cell.Controller
        BufferFull \instance-of BOOLEAN  \derived-from Capacity, Content ->
Cell.Controller, Cell.Foreman
        | END_STATE_COMPONENTS
        | BASIC_CONSTRAINTS
        %DERIVED COMPONENTS
        %INITIAL VALUATION
        | DECLARATIVE CONSTRAINTS
        %STATE BEHAVIOUR
        %ACTION COMPOSITION
        %ACTION DURATION
        | OPERATIONAL CONSTRAINTS
        %PRECONDITIONS
        %EFFECTS OF ACTIONS
        %TRIGGERINGS
        | COOPERATION CONSTRAINTS
        %ACTION PERCEPTION
        %STATE PERCEPTION
        %ACTION INFORMATION
        %STATE INFORMATION
        | END_CONSTRAINTS
      | END_AGENT
| END_SPEC
```

Specification of the society cell

A) Dictionary

The dictionary related to the society contains the following elements:

| block identifier: unique identifier that appears on each block enabling to differentiate it from
| any other block.
BLOCK_ID

| piston: the main part of a piston that bear the rings and on which a rod is added.
PISTON

| cam follower: stick that commands the opening of a valve of the engine. It is placed between
| the camshaft and the valve and it follows the movements of the camshaft to push the valves
| open or closed.
CAM_FOLLOWER

| block: a block with crankshaft, a block with valves or a finished block.
BLOCK = UNION[BLOCK_CRANKSHAFT , BLOCK_VALVE , FINISHED_BLOCK]

| bare block: block on which no parts have been added. It is a triple composed of a block
| identifier, a bearings on checker and a block checked status.
BARE_BLOCK = CP[Id : BLOCK_ID , BearingsOn : BOOLEAN , Checked : ENUM[not , good , bad]]

| bearings on checker : a block have had or not bearings for the camshaft.
BARE_BLOCK_BASIC = BOOLEAN

| bloc checked status: a block is either checked or not. If checked, it is OK or not OK.
BARE_BLOCK_ENUM = ENUM [not, good, bad]

| block with crankshaft: block with pistons that additionally has a crankshaft. The crankshaft
| has been screwed. If the screwing ended successfully, then screwedok is true. Otherwise, it is
| not.
BLOCK_CRANKSHAFT = CP[Block : BLOCK_PISTON , Crank : CRANKSHAFT , ScrewedOk : BOOLEAN]

| block with pistons: couple composed of a bar block and a set of finished pistons.
BLOCK_PISTON = CP[Block : BARE_BLOCK , Pistons : SET[ROD_PISTON_AGGREGATE]]

| finished piston: piston ready to be assembled on the engine block. It is a triple composed of a piston, a set of rings (three), a rod and a peg.
ROD_PISTON_AGGREGATE = CP[Piston : PISTON, Ring : SET[RING], Rod : ROD, Peg : PEG]
| blocks list: sequence of block identifiers.
BLOCK_IDS = SEQ[BLOCK_ID]

| valve status: raw or processed.
STATUS = ENUM[raw , processed]

| cam follower pallet : set of cam followers.

CAM_FOLLOWERS_PALLET = SET[CAM_FOLLOWER]

| blocks stock: list of couples. Each couple is composed of a block identifier and the corresponding block.
BLOCK_STOCK = TABLE[BLOCK_ID --> BLOCK]

| optional block: a possibly absent block
U_BLOCK = BLOCK*

B) Operations

| Removal: remove a set of cam followers from a cam follower pallet.
Removal : CAM_FOLLOWERS_PALLET X SET[CAM_FOLLOWER] -->
CAM_FOLLOWERS_PALLET

C) Composition of the society CELL

1) Agent foreman of the cell :

| the foreman is responsible for the good working of the cell. He is in charge of managing the
| workers and the equipment of the cell and to fix the daily workload by selecting a subset of
| the company plan.

1.A) State components

the foreman of the cell doesn't have any state component.

1.B) Actions

The foreman of the cell can perform the following actions:

| select in the factory plan (a list of block identifiers) to be produced today. This action can by
| perceived by the cell controller.
DetermineDailyPlan (BLOCK_IDS) -> Cell.Conroller

2) Agent cell controller :

| The cell controller is responsible for monitoring the engine production process and to ensure
| the sequencing of operations by sending orders to the machines and operators of the cell. It
| also takes care for the transport of parts and blocks within the cell.

2.A) State components

The state of the cell controller is specified by the following components:

| daily plan for finished blocks : the plan of finished engines to be produced today. It is a
| sequence of block identifiers. This component can be seen by the foreman of the cell.
Finished : Sequence of BLOCK_ID --> Cell.Foreman

| daily plan for semi-finished blocks. the plan of finished engines to be produced today. It is a
| sequence of block identifiers. This component can be seen by the foreman of the cell.
Semi_finished : Sequence of BLOCK_ID --> Cell.Foreman

2.B) Actions

The controller can perform the following actions:

| request the transportation of a block. This action can by perceived by the auto-guided vehicle
| of the blocks transport system. This action is always performed in combination with some
| other actions (see action composition).
*RequestTransport (BLOCK) -> Cell.Blocks_Transport_System.AGV

3) Sub-Society blocks transport system

1) Agent auto-guided vehicle

| an auto-guided vehicle that is able to transport block pallets between a set of given locations.
| The transports are performed on the basis of orders received from the controller.

1.A) State components

The state of the auto-guided vehicle is specified by the following components:

| transported block : this component can be seen by the foreman of the cell.
Transported : instance of BLOCK --> Cell.Foreman

1.B) Actions

The auto-guided vehicle can perform the following actions:

| Load a block. This action is always performed in combination with some other actions
*Load (BLOCK)

| Transport a block . This action is always performed in combination with some other actions
*Transport (BLOCK)

2) Agent buffer

| a location where a blocks can be temporarily put down so that an AGV can be used for
| another operation with higher priority.

2.A) State components

The state of the buffer is specified by the following components:

| buffer content : set of blocks. This component can be seen by the controller of the cell.
Content : Set of BLOCK --> Cell.Controller

| buffer capacity : maximum number of blocks in the buffer. This component has a constant
| value, and it can be seen by the controller of the cell.
*Capacity : Instance of INTEGER --> Cell.Controller

| buffer status : true or false. It is derived from the buffer capacity and buffer content, and it can
| be seen by the controller of the cell and foreman of the cell.
BufferFull : Instance of BOOLEAN Derived from Capacity, Content --> Cell.Controller,
Cell.Foreman

2.B) Actions

the buffer doesn't perform any action.

Annex B

The parser grammar

```
//-----------------------------------------------------------------
// Syntax for paraphrasing data types, operations, agent state components and actions of an
// Albert-II specification.
//-----------------------------------------------------------------

%expression Main
'[ \t\n]+'                              %ignore;
'\|.*\n'                                OneLine;
'\| END_BASIC_TYPES\n'                  %ignore;
'\| END_CONSTRUCTED_TYPES\n'            %ignore;
'\| END_OPERATIONS\n'                   %ignore;
'\| END_SOCIETY\n'                      %ignore;
'\| END_STATE_COMPONENTS\n'             %ignore;
'\| END_ACTIONS\n'                      %ignore;
'\| BASIC_CONSTRAINTS\n'                %ignore;
'\| END_AGENT\n'                        %ignore;
'\| END_CONSTRAINTS\n'                  %ignore;
'\| END_SPEC\n'                         %ignore;
'%DERIVED COMPONENTS'                   %ignore, %push IgnoredPart;
'%SPEC'                                 SpecStart;
'%BASIC TYPES'                          BasicTypes;
'%CONSTRUCTED TYPES'                    ConsTypes;
'%OPERATIONS'                           Ops;
'%SOCIETY'                              Society;
'%AGENT'                                Agent;
'%STATE COMPONENTS'                     StateComp;
'%ACTIONS'                              Actions;
'[0-9]+'                                Number ;

//---------------------------------------------------------------------
//  Keywords
//---------------------------------------------------------------------

'[A-Za-z][A-Za-z0-9_]* | [A-Za-z][A-Za-z0-9_]*\*'  Identifier ;
'BAG | [bB]ag'                                      Bag ;
'CP | cp'                                           CP ;
'ENUM | [eE]num'                                    Enum ;
'SEQ | [sS]eq'                                      Seq ;
'SET | [sS]et'                                      Set ;
'TABLE | [tT]able'                                  Table ;
'UNION | [uU]nion'                                  Union ;
'\\instance\-of'                                    InstanceComp;
'\\set\-of'                                         SetComp;
'\\table\-of'                                       TableComp;
'\\sequence\-of'                                    SeqComp;
'\\indexed\-by'                                     Index;
'\\derived\-from'                                   Derived;
'\\[\*xX]'                                          CartProd;
'='                                                 Equal,        '=' ;
'\*'                                                Star,        '*' ;
'\{'                                                LeftBrace,    '{' ;
'\}'                                                RightBrace,   '}' ;
'\['                                                LeftBracket,  '[' ;
```

```
'\]'                                      RightBracket,  ']' ;
':'                                       Column,              ':' ;
','                                       Comma,               ',' ;
'\('                                      LeftParen,     '(' ;
'\)'                                      RightParen,    ')' ;
'\.'                                      Dot,           '.' ;
'\->'                                     RightArrow ;

%expression IgnoredPart
'.'                                       %ignore;
'\n'                                      %ignore;
'\| END_AGENT'                            %ignore, %pop;

%production spec

Start            spec        -> SpecStart Identifier TypesOpsDec RootSoc;

OneLineComm      Comment     -> OneLine;
MultiLineCom     Comment     -> OneLine Comment;

// Types and Operations Declaration

TypesOpsDec      TypesOpsDec -> BasicDec ConstDec OpsDec;

// Basic Types

BasicVide        BasicDec    -> BasicTypes;
BasicDec         BasicDec    -> BasicTypes BTypeList;
BasicTypeNo      BasicType   -> Identifier;
BasicType        BasicType   -> Comment Identifier;
BType            BTypeList   -> BasicType;
BTypeList        BTypeList   -> BasicType BTypeList;

// Constructed Types

ConstVide        ConstDec    -> ConsTypes;
ConstDec         ConstDec    -> ConsTypes CTypeList;
CType            CTypeList   -> ConstType;
CTypeList        CTypeList   -> ConstType CTypeList;
ConstTypeNo      ConstType   -> Identifier Equal TypeExpr;
ConstType        ConstType   -> Comment Identifier Equal TypeExpr;
UdefExpr         TypeExpr    -> Identifier;
CpExpr           TypeExpr    -> CP LeftBracket Selector RightBracket;
SetExpr          TypeExpr    -> Set LeftBracket TypeExpr RightBracket;
SeqExpr          TypeExpr    -> Seq LeftBracket TypeExpr RightBracket;
BagExpr          TypeExpr    -> Bag LeftBracket TypeExpr RightBracket;
TabExpr          TypeExpr    -> Table LeftBracket TypeExpr RightArrow TypeExpr
RightBracket;
UnExpr           TypeExpr    -> Union LeftBracket TypeList RightBracket;
EnumExpr         TypeExpr    -> Enum LeftBracket TypeList RightBracket;
TypeSing         TypeList    -> TypeExpr;
TypeList         TypeList    -> TypeExpr Comma TypeList;
```

```
Selector            Selector      -> Identifier Column TypeExpr;
SelectorList        Selector      -> Identifier Column TypeExpr Comma Selector;

// Operations

OpsVide             OpsDec        -> Ops;
OpsDecls            OpsDec        -> Ops OpsList;
OneOp               OpsList       -> OpDec;
OpsList             OpsList       -> OpDec OpsList;
OpDeclNo            OpDec         -> Identifier Column OpArity;
OpDecl              OpDec         -> Comment Identifier Column OpArity;
OpArityNoDom        OpArity       -> RightArrow TypeExpr;
OpArityDom          OpArity       -> OpDom RightArrow TypeExpr;
OpDom               OpDom         -> TypeExpr;
OpDoms              OpDom         -> TypeExpr CartProd OpDom;

// Society

RootSocDec          RootSoc       -> SocList;
OneSociety          SocList       -> SocDec AgentList;
SocietyList         SocList       -> SocDec AgentList SocList;
Society             SocDec        -> Society SocAgentName SubSocAgList;
SubSocAg            SubSocAgList  -> SubSocAg;
SubSocAgList        SubSocAgList  -> SubSocAg SubSocAgList;
SubSocAgSing        SubSocAg      -> LeftParen Identifier RightParen;
SubSocAgMult        SubSocAg      -> LeftParen Identifier RightParen RightParen RightParen;

// Agent

OneAgent            AgentList     -> AgentDec;
AgentList           AgentList     -> AgentDec AgentList;
Agent               AgentDec      -> Agent SocAgentName StateCompDec ActionsDec;
RootSocName         SocAgentName-> Identifier;
SubSocAgentName     SocAgentName-> Identifier Dot SocAgentName;

// State Components

NoStateComp         StateCompDec-> StateComp;
StateCompsList      StateCompDec-> StateComp CompList;
CompListEnd         CompList      -> CompDec;
CompList            CompList      -> CompDec CompList;
FixedCompNo         CompDec       -> Star Comp Derivs Exports;
VarCompNo           CompDec       -> Comp Derivs Exports;
FixedComp           CompDec       -> Comment Star Comp Derivs Exports;
VarComp             CompDec       -> Comment Comp Derivs Exports;
NoDerivation        Derivs        -> ;
Derivation          Derivs        -> Derived CompNameList;
OneCompName         CompNameList-> SocAgentName;
ManyCompNames       CompNameList-> SocAgentName Comma CompNameList;
NoExportation       Exports       -> ;
Exportation         Exports       -> RightArrow CompNameList;
SetSeqInstComp      Comp          -> Identifier CompType Identifier;
TableComp           Comp          -> Identifier TableComp Identifier Index Identifier;
```

```
SetComp             CompType     -> SetComp;
SeqComp             CompType     -> SeqComp;
InstanceComp        CompType     -> InstanceComp;

// Actions

ActionNul           ActionsDec   -> Actions ;
ActionDecl          ActionsDec   -> Actions ActionList;

OneAction           ActionList   -> ActionDec;
ActionList          ActionList   -> ActionDec ActionList;

StarActionDecNo     ActionDec    -> Star Identifier ActionBody Exports;
ActionDecNo         ActionDec    -> Identifier ActionBody Exports;
StarActionDec       ActionDec    -> Comment Star Identifier ActionBody Exports;
ActionDec           ActionDec    -> Comment Identifier ActionBody Exports;

ActionNoParams      ActionBody   -> ;
ActionParams        ActionBody   -> '(' TypeList ')';
```

Annex C

The Paraphraser main units

(Globs,Main and Parse)

```
unit Globs;

interface
Uses Classes,Outline, SysUtils;
type

{General  definition}
MyObject=record  { Why not St:^MyRecord?: easy to manipulate }
        Name,U_Name,Plu,U_Def,Body,Man:string;
        TheAn:string[3];
        IsPlu,Star,Undir,Manual,Auto:Boolean;
        case typ:integer of
          1,15:(Rd:String);
          2,3,4,5,6,7:(Value:TList);         {Types:Seq,Set,Enum,Union,Cp,Bag}
          8,9:(Dom,Codom:TList);               {Operations and Table types }
          10:(SocAgents,Agents:TList);         {Societies}
          11:(StateComps,Actions:TList);{Agents}
          12:(Fix:Boolean;CompType:integer;OfType1,OfType2:String;
             Deriv,Exprt:TStringList);       {State Components}
          14:(Args,AExprt:TStringList);        {Actions}
        end;

PMyObject=^MyObject;

Spec= record
     BTypes,CTypes,Operations,Societies:TList;
     end;

var
   MySpec,MySpec1    : Spec;
   PE                : PMyObject;
   Lst               : TList;
   Def_Para,AutoP,ManP,KeepSpec,KeepSem,OkGen,SaveOk,TextAsc,ImpBasic,ImpCons,ImpOp: Boolean;
   FName,Path,ImportName:TFileName;

implementation
end.
```

```
unit Main;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Menus, StdCtrls, Import, Aide,
  Globs, Parser, Wait, Outtype, Prev, DB, DBTables, ExtCtrls, Tabs, Grids, Outline, Buttons,
  VBXCtrl, Chart2fx;

type
  TMainWin = class(TForm)
    OpenDialog1: TOpenDialog;
    Table1: TTable;
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    OpenParaphrase1: TMenuItem;
    Exit1: TMenuItem;
    Panel1: TPanel;
    N1: TMenuItem;
    New1: TMenuItem;
    Save1: TMenuItem;
    Saveas1: TMenuItem;
    Heads: TListBox;
    Close1: TMenuItem;
    Panel2: TPanel;
    OpenBt: TSpeedButton;
    New: TSpeedButton;
    SaveBt: TSpeedButton;
    MainArb: TGroupBox;
    arb: TOutline;
    NewBt: TSpeedButton;
    SaveDialog1: TSaveDialog;
    Paraphrase1: TMenuItem;
    GenerateText1: TMenuItem;
    Import1: TMenuItem;
    ImpBt: TSpeedButton;
    Dictionary1: TMenuItem;
    ObjectBox: TGroupBox;
    OName: TLabel;
    SDef: TMemo;
    PBox: TGroupBox;
    Ex: TMemo;
    Man: TMemo;
    GroupBox2: TGroupBox;
    LName: TLabel;
    LPlu: TLabel;
    Plu: TEdit;
    LDef: TLabel;
    UDef: TEdit;
    TheAn: TComboBox;
    ATM: TBitBtn;
    GoP: TBitBtn;
    Use1: TMenuItem;
    UName: TComboBox;
    Table2: TTable;
    ActBox: TPanel;
    Label1: TLabel;
    Memo1: TMemo;
    Types: TListBox;
    Query1: TQuery;
    DicName: TLabel;
    Auto: TCheckBox;
```

```
    Manual: TCheckBox;
    SemOk: TRadioButton;
    SemKo: TRadioButton;
    Help1: TMenuItem;
    procedure New1Click(Sender: TObject);
    procedure OpenParaphrase1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure arbClick(Sender: TObject);
    procedure NewBtClick(Sender: TObject);
    procedure OpenBtClick(Sender: TObject);
    procedure SaveBtClick(Sender: TObject);
    procedure Save1Click(Sender: TObject);
    procedure GenerateText1Click(Sender: TObject);
    procedure PluKeyUp(Sender: TObject; var Key: Word; Shift: TShiftState);
    procedure ManKeyUp(Sender: TObject; var Key: Word; Shift: TShiftState);
    procedure Close1Click(Sender: TObject);
    procedure Saveas1Click(Sender: TObject);
    procedure PluEnter(Sender: TObject);
    procedure Exit1Click(Sender: TObject);
    procedure Import1Click(Sender: TObject);
    procedure ImpBtClick(Sender: TObject);
    procedure UDefKeyUp(Sender: TObject; var Key: Word; Shift: TShiftState);
    procedure ATMClick(Sender: TObject);
    procedure TheAnChange(Sender: TObject);
    procedure GoPClick(Sender: TObject);
    procedure Memo1Change(Sender: TObject);
    procedure TypesDblClick(Sender: TObject);
    procedure ExChange(Sender: TObject);
    procedure Use1Click(Sender: TObject);
    procedure UNameChange(Sender: TObject);
    procedure AutoClick(Sender: TObject);
    procedure ManualClick(Sender: TObject);
    procedure SemOkClick(Sender: TObject);
    procedure SemKoClick(Sender: TObject);
    procedure ManChange(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  MainWin: TMainWin;
  i,j,k,Sc,Ac,Stc,LastIndx,Counter,SocCount,AgentCount:integer;
  ArbItem,P,P1,P2,Pt:PMyObject;
  CloseOk,modified,NewFile:boolean;
  s,s1,s2,MainSoc,HtmPath:String;
  TypesList:TStrings;

implementation

{$R *.DFM}

{--------------------------------------------------------------------------}
function Plural(O:MyObject):string;
var
  s:string;
begin
  if O.U_Name<>'' then
    s:=O.U_Name
  else
    if O.Name[Length(O.Name)]='*' then
      s:='optional '+LowerCase(Copy(O.Name,1,Length(O.Name)-1))
```

```
    else
      s:=LowerCase(O.Name);
  if O.Plu<>'' then
    Plural:=O.Plu
  else if s[Length(s)]='y' then
    Plural:=Copy(s,1,Length(s)-1)+'ies'
  else if s[Length(s)]<>'s' then
    Plural:=s+'s'
  else
    Plural:=s
end;

{--------------------------------------------------------------------------}
function FindType(Item:string;Level:integer):PMyObject;
var
  i,j,k:integer;
begin
  with MainWin.arb do
  begin
    i:=GetTextItem('BASIC TYPES');
    j:=GetTextItem('OPERATIONS');
    k:=GetTextItem(Item);
    if (k>i) and (k<j) and (Items[k].Level=Level) then
      FindType:=Items[k].Data
    else
      FindType:=nil
  end
end;

{--------------------------------------------------------------------------}
function ObjectName(P:PMyObject):string;
begin
  if P^.U_Name<>'' then
    ObjectName:=P^.U_Name
  else
  begin
    s:=LowerCase(P^.Name);
    if s[Length(s)]='*' then
      s:='optional '+Copy(s,1,Length(s)-1);
    ObjectName:=s
  end
end;

{--------------------------------------------------------------------------}
function Header(Obj:PMyObject):string;
begin
  with Obj^do
  begin
    s:=Name+': ';
    if KeepSem then
    begin
      if (U_Def<>'') and (typ<>0) then s:=ObjectName(obj)+': '+U_Def+'. '+ObjectName(obj)+' is '
      else if U_Def='' then s:=ObjectName(obj)+': '
      else s:=ObjectName(obj)+': '+U_Def+'  '
    end;
    if typ in [0,16] then s:=copy(s,1,length(s)-2)
  end;
  Header:=s
end;

{--------------------------------------------------------------------------}
function ObjectTheAn(s:string):string;
begin
```

```
  if s[1] in ['a','A','i','I','o','O','e','E'] then ObjectTheAn:='an'
  else ObjectTheAn:='a';
end;

{--------------------------------------------------------------------------}
function ParaphraseRedef(AnObject:PMyObject):string;
var
  s,s1:string;
begin
  s:='';
  with AnObject^ do
  begin
    if not KeepSem then
      s:=Name+': a variable of this type can have a value of type '+Rd
    else
    begin
      s:=Header(AnObject);
      if Rd[Length(Rd)]='*' then
      begin
        s1:=Copy(Rd,1,Length(Rd)-1); s:=s+'an optional ';
      end
      else s1:=Rd;
      P:=FindType(s1,2);
      if p<>nil then s:=s+ObjectName(P)
      else s:=s+s1
    end
  end;
  ParaphraseRedef:=s
end;

{--------------------------------------------------------------------------}
function ParaphraseType(AnObject:PMyObject;n:integer;Many:boolean):string;
{n: type level,1 or >1}
 var
  i:integer;
  s,s1:string;
begin
  s:='';s1:='';
  with AnObject^ do
    Case Typ of
    2..4:begin
          case typ of
          2:s1:='set';
          3:s1:='sequence';
          4:if KeepSem then s1:='multi-set'
            else s1:='bag'
          end;
          if KeepSem then
          begin
            if n=1 then s:=Header(AnObject)+'a '+s1
            else if Many then s:=s1+'s'
            else s:=s1;
            s:=s+' of '; Pt:=Value.Items[0];
            if Star then s:=s+' optional ';
            if Undir then s:=s+Plural(Pt^)
            else
            case Pt^.Typ of
              0:if not Pt^.auto then
                 begin
                   if Pt^.Name='INTEGER' then s:=s+' numbers'
                   else if Pt^.Name='BOOLEAN' then s:=s+'true or false values'
                   else if Pt^.Name='STRING' then s:=s+'words'
                   else s:=s+Plural(Pt^)
```

```
          end
          else s:=s+Plural(Pt^);
        else if not Pt^.auto then s:=s+ParaphraseType(Pt,2,true)
        else s:=s+Plural(Pt^);
      end
    end
    else
    begin
      if n=1 then s:=Name+': a '+s1
      else if Many then s:=s1+'s'
      else s:=s1;
      s:=s+' of '; Pt:=Value.Items[0];
      if Undir or (Pt^.Typ in[0,16]) then
      begin
        s:=s+'values of type '+Pt^.Name;
        if Star then s:=s+'*'
      end
      else s:=s+ParaphraseType(Pt,2,true);
    end
  end;
5:begin
  case Value.Count of
    2:s1:='couple';
    3:s1:='triple';
    4:s1:='quadriple';
    else s1:='tuple';
  end;
  if KeepSem then
  begin
    if n=1 then s:=Header(AnObject)+'a '+s1
    else if Many then s:=s1+'s'
    else s:=s1;
    if Many then s:=s+'. Each '+s1+' is composed of '
    else s:=s+' composed of ';
    for i:= 0 to Value.Count-1 do
    begin
      Pt:=Value.Items[i];
      if Pt^.Typ=15 then s:=s+Pt^.TheAn+' '+ObjectName(Pt)+', '
      else if Pt^.Auto then s:=s+Pt^.TheAn+' '+ObjectName(Pt)+', '
      else s:=s+ParaphraseType(Pt,2,False)+', '
    end;
  end
  {else
    begin
    P2:=FindType(Pt^.Name,2);
      s:=s+Pt^.TheAn+' '+ObjectName(Pt)+', '
    else
    begin
      s1:=UpperCase(Pt^.Name[1])+LowerCase(Copy(Pt^.Name,2,Length(Pt^.Name)-1));
      if p2<>nil then
        s:=s+'the field '+s1+' of type '+Pt^.Name+', '
      else
        s:=s+'the field '+s1+' defined as a '+ParaphraseType(Pt,2,False)+', '
    end
  end
  end;}
  ;s:=Copy(s,1,Length(s)-2);
  i:=Length(s);
  while (i>0) and (s[i]<>',') do
    i:=i-1;
  Delete(s,i,1);
  Insert(' and',s,i);
end;
```

```
  6:begin
    if KeepSem then
      s:=ObjectName(AnObject)+': '
    else
      s:='a variable of this type is';
    for i:= 0 to Value.Count-1 do
    begin
      Pt:=Value.Items[i];
      if KeepSem then s:=s+' '+ObjectName(Pt)+' or'
      else s:=s+' of type '+Pt^.Name+' or'
    end;
    s:=copy(s,1,length(s)-3);
   end;
  7:with AnObject^ do
   begin
    if KeepSem then
    begin
      s:=Header(AnObject);
      for i:= 0 to Value.Count-1 do
      begin
        Pt:=Value.Items[i]; s:=s+Pt^.Name+' or '
      end;
      s:=copy(s,1,length(s)-4)
    end
    else
    begin
      s:=Name+' is defined as one of the following constant values: ';
      for i:= 0 to Value.Count-2 do
      begin
        Pt:=Value.Items[i]; s:=s+Pt^.Name+', '
      end;
      s:=copy(s,1,length(s)-2); Pt:=Value.Items[i+1]; s:=s+' or '+Pt^.Name;
    end;
   end;
  8:begin
    if n=1 then if KeepSem then
    begin
      s:=ObjectName(AnObject);
      if U_Def<>'' then s:=s+'(U_Def)';
      s:=s+': '
    end
    else
    begin
      s:=Header(AnObject);
      if U_Def='' then s:=s+' defined as a '
      else s:=s+'. It is defined as a '
    end;
    if KeepSem then
      s:=s+'list of couples. Each couple is composed of '
    else
      s:=s+'table indexed by ';
    Pt:=Dom.Items[0];
    if KeepSem then
      s:=s+Pt^.TheAn+' '+ObjectName(Pt)+' and the corresponding '
    else          s:=s+'values of type '+Pt^.Name+' and containes ';
    P:=Codom.Items[0];
    if KeepSem then  s:=s+ObjectName(P)
    else          s:=s+'values of type '+P^.Name
   end;
  end;
  ParaphraseType:=s
end;
```

```
{-----------------------------------------------------------------------}
function ParaObjectName(Text:String; Many:boolean):string;
var
 s,s1:string;
begin
 s:=Text;s1:='';
 if s[Length(s)]='*' then
 begin
  s:=Copy(s,1,Length(s)-1);s1:=' (if exists) '
 end;
 P:=FindType(s,2);
 if p<>nil then
 begin
  if Many then ParaObjectName:=Plural(P^)+s1
  else  ParaObjectName:=ObjectName(P)+s1
 end
 else
  ParaObjectName:=LowerCase(s)+s1
end;

{-----------------------------------------------------------------------}
function ParaphraseOpAct(Obj:PMyObject):string;
var
 Many:Boolean;
 s:string;
 k:integer;
begin
 s:=Obj^.Name;
 if s[1]='*' then s:=Copy(s,2,Length(s)-1);
 if (Obj^.U_Def='') and (Obj^.Typ=9) then
 begin
  with Obj^ do
  begin
   s:=s+': determins ';
   P:=Codom.Items[0];
   if not KeepSem then s:=s+P^.Name+' from '
   else s:=s+P^.TheAn+' '+ParaObjectName(P^.Name,False)+' from ';
   for i:= 0 to Dom.Count-2 do
   begin
    P:=Dom.Items[i];
    if KeepSem then s:=s+P^.TheAn+' '+ObjectName(P)+', '
    else s:=s+P^.Name+', '
   end;
   P:=Dom.Items[Dom.Count-1];
   if Dom.Count>1 then
   begin
    s:=copy(s,1,length(s)-2); s:=s+' and ';
   end;
   if not KeepSem then s:=s+P^.Name
   else s:=s+P^.TheAn+' '+ObjectName(P)
  end
 end
 else if Obj^.U_Def<>'' then
 begin
  s:=s+': '+Obj^.U_Def;
  with Obj^ do
  begin
   for i:=0 to MainWin.Types.Items.Count-1 do
   begin
    s1:=MainWin.Types.Items[i];
    j:=Pos(s1,s);
    if j>0 then
    begin
```

```
        if s[j+Length(s1)]='*' then
          s1:=s1+'*';
        if s[j+Length(s1)]='s' then
          s1:=s1+'s';
        Many:=s1[Length(s1)]='s';
        Delete(s,j,Length(s1));
        Insert(ParaObjectName(s1,Many),s,j)
      end
    end;
   end;
  end;
  if (Obj^.Typ=14)then with Obj^ do
  begin
   if Name[1]='*' then
    s:=s+'. This action is allways performed in combination with some other actions';
   if (AExprt<>nil)and (AExprt.Count>0)then with AExprt do
   begin
    if Name[1]='*' then s:=s+' and it is excerced on '
    else s:=s+'. This action is excerced on ';
    for i:=0 to Count-1 do
    begin
     k:=Pos('.',Strings[i]);
     s:=s+'the '+LowerCase(Copy(Strings[i],k+1,Length(Strings[i])-k))+', ';
    end;
    s:=Copy(s,1,Length(s)-2)
   end;
  end;
  ParaphraseOpAct:=s
end;

{-------------------------------------------------------------------}
function ParaphraseSociety(AnObject:PMyObject):string;
var
  i,j:integer;
  s,s1,UN:string;
begin
  with AnObject^ do
  begin
   s:=ObjectName(AnObject);
   if U_Def<>'' then
    s:=s+'(U_Def)';
   s:=s+': composed of ';
   for i:=0 to Agents.Count-1 do
   begin
    Pt:=Agents.Items[i];
    if Pt^.IsPlu then
     s:=s+'several '+Plural(Pt^)+', '
    else
     s:=s+Pt^.TheAn+' '+ObjectName(Pt)+', '
   end;
  end;
  s:=Copy(s,1,Length(s)-2);
  j:=Length(s);
  while (j>1) and (s[j]<>',') do
   j:=j-1;
  Delete(s,j,2);
  Insert(' and ', s, j);
  ParaphraseSociety:=s;
end;

{-------------------------------------------------------------------------}
function ParaphraseComp(AnObject:PMyObject):string;
var
```

```
 s,s1:string;
begin
 s:='';
 with AnObject^ do
 begin
  if not KeepSem then
  begin
   s:=Header(AnObject);
   if U_Def<>'' then
    s:=s+' '+(U_Def)+'. It is defined as '
   else
    s:=s+' defined as ';
   case CompType of
   1:s:=s+'a set of values of type '+OfType1;
   2:s:=s+'a sequence of values of type '+OfType1;
   3:s:=s+'a single value of type '+OfType1;
   4:s:=s+'a table indexed by values of type '+OfType1+' and containes values of type '+OfType2
   end
  end
  else
  begin
   s:=ObjectName(AnObject);
   if U_Def<>'' then
    s:=s+': '+U_Def+'. This component is '
   else
    s:=s+' witch is ';
   case CompType of
   1:s:=s+'a set of ';
   2:s:=s+'a sequence of ';
   4:s:=s+'a list of couples of ';
   end;
   s1:=OfType1;
   if s1[Length(s1)]='*' then
   begin
    s1:=Copy(s1,1,Length(s1)-1); Star:=True;
   end;
   P:=FindType(s1,2);
   if P<>nil then
   begin
    if CompType<>3 then
    begin
     if star then s:=s+' optional '+Plural(P^)
     else s:=s+Plural(P^)
    end
    else if star then s:=s+'an optional '+ObjectName(P)
       else s:=s+P^.TheAn+' '+ObjectName(P)
   end
   else
    if OfType1='BOOLEAN' then s:=s+' true or false'
    else s:=s+' '+LowerCase(OfType1);
  end;
  if CompType=4 then
  begin
   s1:=OfType2;
   if s1[Length(s1)]='*' then
   begin
    s1:=Copy(s1,1,Length(s1)-1); Star:=True;
   end;
   s:=s+' and ';
   P:=FindType(s1,2);
   if P<>nil then
   begin
    if star then s:=s+' optional '+Plural(P^)
```

```
        else s:=s+Plural(P^)
      end
      else s:=s+' '+LowerCase(OfType2)+'s';
    end;
    if Fix then s:=s+'. This component has a constant value';
    if (Deriv<>nil) and (Deriv.Count>0) then
    begin
      if Fix then  s:=s+' and is derived from the component'
      else         s:=s+'. It is derived from the component';
      if Deriv.Count>1 then s:=s+'s ';
      for i:= 0 to Deriv.Count-2 do
        s:=s+ Deriv.Strings[i]+',';
      s:=copy(s,1,length(s)-1);
      if Deriv.Count>1 then  s:=s+' and '+Deriv.Strings[i+1];
    end;
    if (Exprt<>nil) and (Exprt.Count>0) then
    begin
      k:=Pos('.',Exprt.Strings[0]);
      j:=Length(Exprt.Strings[0]);
      if (Deriv=nil) and (not Fix) then s:=s+'. This component can be seen by the '
      else  s:=s+', and it can be seen by the ';
      if Exprt.Count=1 then   s:=s+Copy(LowerCase(Exprt.Strings[0]),k+1,j-k)
      else
      begin
        for i:= 0 to Exprt.Count-2 do
        begin
          k:=Pos('.',Exprt.Strings[i]);
          j:=Length(Exprt.Strings[i]);
          s:=s+Copy(LowerCase(Exprt.Strings[i]),k+1,j-k)+',';
        end;
        s:=copy(s,1,length(s)-1);
        k:=Pos('.',Exprt.Strings[i+1]);
        j:=Length(Exprt.Strings[i+1]);
        s:=s+' and '+Copy(LowerCase(Exprt.Strings[i+1]),k+1,j-k);
      end
    end
  end;
  ParaphraseComp:=s
end;

{----------------------------------------------------------------------}
function Generate(Obj:PMyObject):string;
var
  s,s1:string;
begin
  s1:='';
  with Obj^ do
  begin
    case Typ of
    0,11,16:s:=Header(Obj);
    1:s:=ParaphraseRedef(Obj);
    2..8:s:=ParaphraseType(Obj,1,False);
    {A value of type X is obtained by applying the operation Y on values of type ...}
    9,14:s:=ParaphraseOpAct(Obj);
    10:s:=ParaphraseSociety(Obj);
    12:s:=ParaphraseComp(Obj);
    end;
  end;
  Generate:=s
end;

{---------------------------------------------------------------------------}
procedure ParaphraseObject(Obj:PMyObject);
```

```
var
  s:string;
begin
  s:='';
  if Def_Para then
  begin
    AutoP:=Obj^.Auto;ManP:=Obj^.Manual
  end;
  if AutoP or ManP then with Obj^,Preview.OutText do
  begin
    if not TextAsc then Lines.Add('<li>')
    else s:='| ';
    if AutoP   then s:=s+Generate(Obj);
    if ManP then s:=s+Man;
    Lines.Add(s);
    if not TextAsc then Lines.Add('</li>');
  end;
  if KeepSpec then with Obj^,Preview.OutText do
  begin
    if not TextAsc then Lines.Add('<p>');
    Lines.Add(Body);
    if not TextAsc then Lines.Add('</p>');
  end;
  Preview.OutText.Lines.Add('');
end;

{---------------------------------------------------------------------------}
procedure GenerateAgentText(P:PMyObject; nb:integer);
var
  i,j,k:integer;
  s,s1:string;
begin
  with MainWin.arb, Preview.OutText do
  begin
    ArbItem:=MainWin.arb.Items[GetTextItem('Agent : '+P^.Name)].Data;
    s:=Generate(ArbItem);s1:=IntToStr(nb);
    Lines.Add(s1+') Agent '+UpperCase(ObjectName(P)));Lines.Add('');
    Lines.Add(IntToStr(nb)+'.A) State components');Lines.Add('');
    if P^.StateComps<>nil then
    begin
      Lines.Add('The state of the '+ObjectName(P)+' is specified by the following components:');Lines.Add('');
      for j:= 0 to P^.StateComps.Count-1 do
      begin
        P2:=P^.StateComps.Items[j];
        ArbItem:=MainWin.arb.Items[MainWin.arb.GetTextItem(P2^.Name)].Data;
        ParaphraseObject(ArbItem);
      end;
    end
    else
    begin
      Lines.Add('the '+ObjectName(P)+' dosn''t have any state component');Lines.Add('');
    end;
    Lines.Add(IntToStr(nb)+'.B) Actions');Lines.Add('');
    if P^.Actions<>nil then
    begin
      Lines.Add('The '+ObjectName(P)+' can perfomed the following actions:');Lines.Add('');
      for j:= 0 to P^.Actions.Count-1 do
      begin
        P2:=P^.Actions.Items[j];
        ArbItem:=MainWin.arb.Items[MainWin.arb.GetTextItem(P2^.Name)].Data;
        ParaphraseObject(ArbItem);
      end;
    end
```

```
    else
    begin
      Lines.Add('the '+ObjectName(P)+' dosn''t performe any action');Lines.Add('')
    end
  end;
end;

{----------------------------------------------------------------------------}
procedure GenerateAgentHtm(P:PMyObject;indx:string);
var
  j:integer;
  s:string;
begin
  with MainWin.arb, Preview.OutText do
  begin
    Lines.Clear;
    ArbItem:=MainWin.arb.Items[GetTextItem('Agent : '+P^.Name)].Data;
    Lines.Add('<FONT SIZE="-1">');
    Lines.Add('<HTML><HEAD><TITLE> Albert Paraphraser </TITLE></HEAD>');
    Lines.Add('<body background="Ground.gif" bgcolor="#FFFFFF" bgproperties="fixed">');
    Lines.Add('<H2><CENTER> Agent '+P^.Name+'</CENTER></H2>');
    Lines.Add('<H3>A) State components</H3>');
    if P^.StateComps<>nil then
    begin
      Lines.Add('<p> The state of the '+ObjectName(P)+' is specified by the following components: </p>');
      for j:= 0 to P^.StateComps.Count-1 do
      begin
        P2:=P^.StateComps.Items[j];
        ArbItem:=MainWin.arb.Items[MainWin.arb.GetTextItem(P2^.Name)].Data;
        ParaphraseObject(ArbItem);
      end;
    end
    else
      Lines.Add('<p> The '+ObjectName(P)+' dosn''t have any state component</p>');
    Lines.Add('<H3>B) Actions</H3>');
    if P^.Actions<>nil then
    begin
      Lines.Add('<p> The '+ObjectName(P)+' can perfomed the following actions: </p>');
      for j:= 0 to P^.Actions.Count-1 do
      begin
        P2:=P^.Actions.Items[j];
        ArbItem:=MainWin.arb.Items[MainWin.arb.GetTextItem(P2^.Name)].Data;
        ParaphraseObject(ArbItem);
      end;
    end
    else
      Lines.Add('<p>The '+ObjectName(P)+' dosn''t performe any action</p>');
    Lines.Add('<p><img src="Line.gif" width="600" height="4"></p>');
    s:='<p> <a HREF="file://'+FName+'.htm"> Main Society </a> </p>';
    Lines.Add(s);
    Lines.Add(' </HTML>');
    Lines.SaveToFile('Agent'+indx+'.htm');
  end;
end;

{----------------------------------------------------------------------------}
procedure GenerateSociety(P:PMyObject; pos:integer);
var
  i:integer;
begin
  with MainWin.arb, Preview.OutText do
  begin
    if pos=0 then
```

```
    Lines.Add('C) Composition of the society '+UpperCase(ObjectName(P)))
   else
   begin
    Lines.Add(IntToStr(Pos)+') Sub-Society '+UpperCase(ObjectName(P)));Lines.Add('');
    Lines.Add('Composition of the society')
   end;Lines.Add('');
   for i:=0 to P^.SocAgents.Count-1 do
   begin
    P1:=P^.SocAgents.Items[i]; P2:=P^.Agents.Items[i]; P2^.Name:=P1^.Name;
    if (P2^.Typ<>10)then GenerateAgentText(P2,i+1)
   end
  end
end;

{-----------------------------------------------------------------------------}
procedure GenerateHtmlSubSociety(P:PMyObject;indx:string);
var
 i,j:integer;
 s:string;
begin
 s:=P^.Name;
 with Preview.OutText do
 begin
   Lines.Clear;
   Lines.Add('<FONT SIZE="-1">');
   Lines.Add('<HTML><HEAD><TITLE> Albert Paraphraser </TITLE></HEAD>');
   Lines.Add('<body background="Ground.gif" bgcolor="#FFFFFF" bgproperties="fixed">');
   Lines.Add('<H2><CENTER> Sub-Society '+ObjectName(P)+'</CENTER></H2>');
   Lines.Add('<H3> Society composition </H3>');
   Lines.Add('<ul>');AgentCount:=0;SocCount:=0;
   for j:=0 to P^.SocAgents.Count-1 do
   begin
    P1:=P^.Agents.Items[j];
    if P1^.Typ<>10 then
    begin
     s:='<li> Agent '+'<A HREF="'+HtmPath+'Agent'+Indx+IntToStr(AgentCount)+'.htm">
'+ObjectName(P1)+' </A>,</li>';
     Lines.Add(s);AgentCount:=AgentCount+1
    end
    else
    begin
     s:='<li> Sub-Society '+'<A HREF="'+HtmPath;
     s:=s+'Sub_S'+indx+IntToStr(SocCount)+'.htm"> '+ObjectName(P1)+' </A>,</ul>';
     Lines.Add(s);SocCount:=SocCount+1
    end
   end;
   Lines.Add('</ul> <p><img src="Line.gif" width="600" height="4"></p>');
   s:='<p> <a HREF="file://'+FName+'.htm"> Main Society </a> </p>';
   Lines.Add(s);
   Lines.Add(' </HTML>');
   Lines.SaveToFile('Sub_S'+indx+'.htm');AgentCount:=0;SocCount:=0;
   for i:=0 to P^.Agents.Count-1 do
   begin
    P1:=P^.Agents.Items[i];
    if (P1^.Typ=10) then
    begin GenerateHtmlSubSociety(P1,indx+IntToStr(SocCount));SocCount:=SocCount+1; end
    else begin GenerateAgentHtm(P1,indx+IntToStr(AgentCount));AgentCount:=AgentCount+1;end
   end
 end;
end;

{-----------------------------------------------------------------------------}
procedure GenerateText;
```

```
var
  p,p1,p2:PMyObject;
  i,j,k:integer;
  s,s1:string;
begin
  Counter:=1;
  with MainWin.Arb, MySpec, Preview.OutText do
  begin
    Preview.OutText.Clear;
    p:=Societies.Items[0];
    Lines.Add('                              Specification of the society '+ObjectName(P));
    Lines.Add('');
    Lines.Add('A) Dictionary');Lines.Add('');
    Lines.Add('The dictionary related to the society contains the following elements:');
    Lines.Add('');
    if BTypes<>nil then
    begin
      for i:=0 to BTypes.Count-1 do
      begin
        P:=BTypes.Items[i];
        ArbItem:=Items[GetTextItem(P^.Name)].Data;
        if ArbItem^.Auto or (ArbItem^.Manual and (ArbItem^.Man<>'')) then
          ParaphraseObject(ArbItem);
      end;
    end;
    if CTypes<>nil then
    begin
      for i:=0 to CTypes.Count-1 do
      begin
        P:=CTypes.Items[i];
        ArbItem:=MainWin.Arb.Items[GetTextItem(P^.Name)].Data;
        if ArbItem^.Auto or (ArbItem^.Manual and (ArbItem^.Man<>'')) then
          ParaphraseObject(ArbItem);
      end;
      Lines.Add('');
    end;
    if Operations<>nil then
    begin
      Lines.Add('B) Operations');Lines.Add('');
      if Operations.Count>1 then
      begin
        if KeepSem then
          Lines.Add('The operations defined on the dictionary items are:')
        else
          Lines.Add('The operations defined on the types are:');
        Lines.Add('');
      end;
      for i:=0 to Operations.Count-1 do
      begin
        P:=Operations.Items[i];
        ArbItem:=MainWin.Arb.Items[GetTextItem(P^.Name)].Data;
        if ArbItem^.Auto or (ArbItem^.Manual and (ArbItem^.Man<>'')) then
          ParaphraseObject(ArbItem)
      end;
      Lines.Add('');
    end;
    P:=Societies.Items[0];
    GenerateSociety(P,0);
    for i:=0 to P^.SocAgents.Count-1 do
    begin
      P1:=P^.SocAgents.Items[i];
      P2:=P^.Agents.Items[i]; P2^.Name:=P1^.Name;
      if (P2^.Typ=10) then GenerateSociety(P2,i+1)
```

```
    end
  end
end;

{-------------------------------------------------------------------------}
procedure GenerateHtml;
var
  p,p1:PMyObject;
  i,j,k:integer;
  s:string;
begin
  with MainWin.Arb, MySpec, Preview.OutText do
  begin
    p:=Societies.Items[0];
    Lines.Add('<FONT SIZE="-1">');
    Lines.Add('<HTML><HEAD><TITLE> Albert Paraphraser </TITLE></HEAD>');
    Lines.Add('<body background="Ground.gif" bgcolor="#FFFFFF" bgproperties="fixed">');
    Lines.Add('<H2> <CENTER> Specification of the society '+ UpperCase(ObjectName(P))+' </CENTER>
</H2>');
    Lines.Add('<H3>A) Dictionary </H3>');
    Lines.Add('<p> The dictionary related to the society contains the following items:</p>');
    if BTypes<>nil then
    begin
      for i:=0 to BTypes.Count-1 do
      begin
        P:=BTypes.Items[i];
        ArbItem:=Items[GetTextItem(P^.Name)].Data;
        if ArbItem^.Auto or ArbItem^.Manual then
          ParaphraseObject(ArbItem);
      end;
    end;
    if CTypes<>nil then
    begin
      for i:=0 to CTypes.Count-1 do
      begin
        P:=CTypes.Items[i];
        ArbItem:=MainWin.Arb.Items[GetTextItem(P^.Name)].Data;
        if ArbItem^.Auto or ArbItem^.Manual then
          ParaphraseObject(ArbItem);
      end;
    end;
    if Operations<>nil then
    begin
      Lines.Add('<H3> B) Operations </H3>');
      if Operations.Count>1 then
        Lines.Add('<p> The operations defined on the dictionary items are: </p>')
      else
        Lines.Add('<p> The operation defined on dictionary items is:</p>');
      for i:=0 to Operations.Count-1 do
      begin
        P:=Operations.Items[i];
        ArbItem:=MainWin.Arb.Items[GetTextItem(P^.Name)].Data;
        if ArbItem^.Auto or ArbItem^.Manual then
          ParaphraseObject(ArbItem);
      end;
    end;
    P1:=Societies.Items[0];AgentCount:=0;
    Lines.Add('<H3> C) Society composition </H3>'); Lines.Add('<ul>');
    for i:=0 to P1^.SocAgents.Count-1 do
    begin
      P:=P1^.Agents.Items[i];
      if P^.Typ<>10 then
      begin
```

```
      s:='<li> Agent '+'<A HREF="'+HtmPath+'Agent'+IntToStr(AgentCount)+'.htm"> '+ObjectName(P)+'
</A>,</li>';
      Lines.Add(s);AgentCount:=AgentCount+1;
    end
    else
    begin
      s:='<li> Sub-Society '+'<A HREF="'+HtmPath;
      s:=s+'Sub_S'+IntToStr(SocCount)+'.htm"> '+ObjectName(P)+' </A>,</li>';
      Lines.Add(s);SocCount:=SocCount+1;
    end
  end; Lines.Add('</ul>');Lines.SaveToFile(FName+'.htm');AgentCount:=0;SocCount:=0;
  for i:=0 to P1^.Agents.Count-1 do
  begin
    P2:=P1^.Agents.Items[i];
    if (P2^.Typ=10) then
    begin GenerateHtmlSubSociety(P2,IntToStr(SocCount));SocCount:=SocCount+1;end
    else begin GenerateAgentHtm(P2,IntToStr(AgentCount));AgentCount:=AgentCount+1; end
  end
 end;
end;

{-----------------------------------------------------------------------------}
function TypeName(i:integer):string;
begin
 case i of
  2:TypeName:='TSet';
  3:TypeName:='TSeq';
  4:TypeName:='TBag';
  5:TypeName:='TCp';
  6:TypeName:='TUnion';
  7:TypeName:='TEnum';
  8:TypeName:='TTable';
 end
end;

{-----------------------------------------------------------------------------}
procedure AddType(P:PMyObject;n:integer);
var
 indx,i,j,len:integer;
 P2,P1:PMyObject;
begin
 P^.IsPlu:=True;Pt^.Manual:=False;Pt^.Auto:=True; P^.Man:='';P^.Plu:='';
 P^.U_Name:='';P^.U_Def:='';P^.TheAn:=ObjectTheAn(P^.Name);
 with MainWin.arb do
 begin
  indx:=AddChild(n,P^.Name);Items[indx].Data:=P;
  case p^.Typ of
  2..4:begin
       P1:=P^.Value.Items[0]; s1:=P1^.Name;len:=Length(s1);
       if P1^.Typ=1 then
       begin
        if s1[len]='*' then
        begin
         P^.Star:=True; s1:=Copy(s1,1,len-1);
        end
        else P^.Star:=False;
        P2:=FindType(s1,2);
        if P2<>nil then
        begin
         P^.Value.Items[0]:=P2;P^.Undir:=True
        end
        else
        begin
```

```
        P1^.Typ:=0; AddType(P1,GetTextItem(P^.Name))
       end
      end
      else
      begin
       P1^.Name:=P^.Name+'_'+TypeName(P1^.Typ);
       AddType(P1,GetTextItem(P^.Name))
      end
     end;
  5:for i:= 0 to P^.Value.Count-1 do
    begin
     P1:=P^.Value.Items[i]; P1^.Body:=P^.Name+'.'+P1^.Name+': '+P1^.Body;
     AddType(P1,GetTextItem(P^.Name))
    end;
  6:for i:= 0 to P^.Value.Count-1 do
    begin
     P1:=P^.Value.Items[i];{P1^.Name:=P^.Name+'_'+IntToStr(i+1);}
     if P1^.Typ=1 then
     begin
      P2:=FindType(P1^.Name,2);
      if P2<>nil then
       P^.Value.Items[i]:=P2
      else
      begin
       P1^.Typ:=0; AddType(P1,GetTextItem(P^.Name))
      end
     end
     else
      AddType(P1,GetTextItem(P^.Name));
    end;
  8:begin
     P1:=P^.Dom.Items[0];
     if P1^.Typ=1 then
     begin
      P2:=FindType(P1^.Name,2);
      if P2<>nil then
       P^.Dom.Items[0]:=P2
      else
      begin
       P1^.Typ:=16; P1^.Body:=P^.Name+'.Dom: '+P1^.Rd;
       AddType(P1,GetTextItem(P^.Name))
      end
     end
     else
      AddType(P1,GetTextItem(P^.Name));
     P1:=P^.Codom.Items[0];
     if P1^.Typ=1 then
     begin
      P2:=FindType(P1^.Name,2);
      if P2<>nil then
       P^.Codom.Items[0]:=P2
      else
      begin
       P1^.Body:=P^.Name+'.Codom: '+P1^.Rd;P1^.Typ:=16;
       AddType(P1,GetTextItem(P^.Name))
      end
     end
     else
      AddType(P1,GetTextItem(P^.Name));
    end;
  end;
 end
end;
```

```
{------------------------------------------------------------------------}
procedure AddSociety(Pt:PMyObject; i:integer);
var
  P,P1,P2:PMyObject;
  n,k,indx:integer;
begin
  with MainWin.arb do
  begin
    if i=1 then
      Sc:=AddChild(GetTextItem(MainSoc),'Society : '+Pt^.Name);
    Items[Sc].Data:=Pt;
    for n:= 0 to Pt^.Agents.Count-1 do
    begin
      P:=Pt^.Agents.Items[n];P1:=Pt^.SocAgents.Items[n];
      P^.IsPlu:=P1^.IsPlu;P^.Name:=P1^.Name;
      P^.Manual:=False;P^.Auto:=True;
      if P^.Agents=nil then
        P^.Body:='Agent';
      Ac:=AddChild(Sc,'Agent : '+P1^.Name);Items[Ac].Data:=P;
      if P^.StateComps<>nil then
      begin
        Stc:=AddChild(Ac,'STATE COMPONENTS');
        New(P2);P2^.Typ:=-1;
        for k:= 0 to P^.StateComps.Count-1 do
        begin
          P1:=P^.StateComps.Items[k];indx:=AddChild(Stc,P1^.Name);
          P1^.Manual:=False;P1^.Auto:=True;P1^.TheAn:='the'; Items[indx].Data:=P1;
        end;
        Items[Stc].Data:=P2;
      end;
      if P^.Actions<>nil then
      begin
        New(P2);P2^.Typ:=-1;
        Stc:=AddChild(Ac,'ACTIONS');
        Items[Stc].Data:=P2;
        for k:= 0 to P^.Actions.Count-1 do
        begin
          P1:=P^.Actions.Items[k];indx:=AddChild(Stc,P1^.Name);
          Items[indx].Data:=P1;P1^.Manual:=False;P1^.Auto:=True;
          if P1^.Args<>nil then with P1^.Args,MainWin.Types do
      {if an argument is of type T* and T* is not in the types list, then add it}
          for j:=0 to P1^.Args.Count-1 do
            if (Strings[j][Length(Strings[j])]='*')and(Items.IndexOf(Strings[j])=-1) then
              Items.Add(Strings[j])
        end;
      end
    end
  end
end;

{------------------------------------------------------------------------
Get all the information found in the specification file and put them in    }
procedure EmptySpec(var arb:TOutline);
var
  i,indx:integer;
  P3:PMyObject;
begin
  arb.Clear;
  arb.Lines.AddStrings(MainWin.Heads.Items);
  arb.SelectedItem:=0;
  with arb do
  begin
```

```
New(P1);P1^.Typ:=-1;
if MySpec.BTypes<>nil then
begin
  Lst:=MySpec.BTypes;
  for i:= 0 to Lst.Count-1 do
  begin
    Pt:=Lst.Items[i];Pt^.Body:=Pt^.Name; MainWin.Types.Items.Add(Pt^.Name);
    AddType(Pt,1)
  end
end;
Items[GetTextItem('BASIC TYPES')].Data:=P1;
New(P1);P1^.Typ:=-1;
if MySpec.CTypes<>nil then
begin
  Lst:=MySpec.CTypes;
  for i:= 0 to Lst.Count-1 do
  begin
    Pt:=Lst.Items[i]; MainWin.Types.Items.Add(Pt^.Name);
    AddType(Pt,GetTextItem('CONSTRUCTED TYPES'))
  end;
end;
Items[GetTextItem('CONSTRUCTED TYPES')].Data:=P1;
New(P1);P1^.Typ:=-1;
if MySpec.Operations<>nil then
begin
  Lst:=MySpec.Operations;
  Items[GetTextItem('OPERATIONS')].Data:=P1;
  for i:= 0 to Lst.Count-1 do
  begin
    Pt:=Lst.Items[i];Pt^.Manual:=False;Pt^.Auto:=True;
    for k:=0 to Pt^.Dom.Count-1 do
    begin
      P2:=Pt^.Dom.Items[k];
  {if an argument is of type T* and T* is not in the types list, then add it}
      with MainWin.Types,P2^ do
        if (Name[Length(Name)]='*')and(Items.IndexOf(Name)=-1) then
          Items.Add(Name);
      if P2^.Typ=1 then
      begin
        P2^.TheAn:=ObjectTheAn(P2^.Name);
        P3:=FindType(P2^.Name,2);
        if P3<>nil then
          Pt^.Dom.Items[k]:=P3
      end;
    end;
    P2:=Pt^.Codom.Items[0];
    with MainWin.Types,P2^ do
      if (Name[Length(Name)]='*')and(Items.IndexOf(Name)=-1) then
        Items.Add(Name);
    if P2^.Typ=1 then
    begin
      P2^.TheAn:=ObjectTheAn(P2^.Name);s1:=P2^.Name;
      if s1[length(s1)]='*' then
      begin
        P2^.Star:=True; s1:=Copy(s1,1,length(s1)-1);
      end
      else P2^.Star:=False;
      P3:=FindType(s1,2);
      if P3<>nil then
      begin
        Pt^.Codom.Items[0]:=P3;
      end
    end;
```

```
        indx:=AddChild(GetTextItem('OPERATIONS'),Pt^.Name);
        Items[indx].Data:=Pt;
      end;
    end;
    Items[GetTextItem('OPERATIONS')].Data:=P1;
    Pt:=MySpec.Societies.Items[0];Pt^.Manual:=False;Pt^.Auto:=True;
    MainSoc:='SOCIETY : '+Pt^.Name;
    Items[GetTextItem('SOCIETIES')].Text:=MainSoc;
    Sc:=GetTextItem(MainSoc);
    AddSociety(Pt,0);
    for i:= 1 to MySpec.Societies.Count-1 do
    begin
      P1:=MySpec.Societies.Items[i];
      P:=Pt^.SocAgents.Items[Pt^.Agents.Count+i-1];
      P1^.IsPlu:=P^.IsPlu;
      P1^.Name:=P^.Name;
      Pt^.Agents.Add(P1);
      AddSociety(P1,1);
    end;
  end;
end;

{-----------------------------------------------------------------------}
procedure CleanDeskTop(State:boolean);
var
  t:boolean;
begin
  t:=not State;
  with MainWin do
  begin
    Panel1.visible:=t;SaveBt.Enabled:=t;Save1.Enabled:=t;
    Close1.Enabled:=t;SaveAs1.Enabled:=t;
    GenerateText1.Enabled:=t;ImpBt.Enabled:=t;Import1.Enabled:=t;
    modified:=t;Use1.Enabled:=t;
    if State then
      MainWin.Caption:='AlbertII Paraphraser'
  end
end;

{-----------------------------------------------------------------------}
procedure CleanObject;
begin
  with MainWin do
  begin
    UName.Text:='';SDef.Clear;UDef.Clear;Ex.Clear;Man.Clear;Plu.Clear;
    TheAn.Text:='';ObjectBox.Caption:='Selected Object '
  end
end;

{-----------------------------------------------------------------------}
function CleanName(FName:TFileName):String;
begin
  i:=length(FName);
  while (i>0) and (FName[i]<>'\') do
    i:=i-1;
  CleanName:=LowerCase(Copy(FName,i+1,length(FName)-i))
end;

{-----------------------------------------------------------------------}
procedure TMainWin.New1Click(Sender: TObject);
var
  SourceName:array[0..79] of Char;
begin
```

```
  OpenDialog1.Filter:='specification files (*.txt)|*.txt|All files (*.*)|*.*';
  OpenDialog1.FileName:=CleanName(FName)+'.txt';
  If OpenDialog1.Execute then
  begin
    if modified and (MessageDlg('File '+CleanName(FName)+' was modified. Save changes?',
      mtInformation, [mbYes, mbNo,mbCancel], 0)=mrYes) then
      Save1Click(Sender);
      s:=OpenDialog1.FileName;
      i:=Pos('.',s);
      if i>0 then
        FName:=Copy(s,1,i-1)
      else
        FName:=s;
      Path:=FName;
      StrPCopy(SourceName,s);
      Messag.Show;
      Messag.Canvas.Brush.Color:=clBtnFace;
      Messag.Canvas.TextOut(72,40,'Analysing Specification File.');
      AnalyseSpec(SourceName);
      Messag.Close;
      EmptySpec(arb);
      CleanDeskTop(False);
      CleanObject;
      NewFile:=True;
      modified:=false;
      MainWin.Caption:='AlbertII Paraphraser - '+CleanName(FName)
  end
end;

{-----------------------------------------------------------------------}
procedure AddInfo(var arb:TOutline; FName:TFileName);
var
  SourceName:array[0..79] of Char;
begin
  with MainWin.table1 do
  begin
    TableName:=FName;Open;First;Path:=FieldByName('Man').AsString;
    StrPCopy(SourceName,path+'.txt');AnalyseSpec(SourceName);EmptySpec(arb);
    Next;
    while not eof do
    begin
      s:=FieldByName('Text').AsString;
      if s[3]='_' then s:=Copy(s,4,Length(s)-3);
      P:=arb.Items[arb.GetTextItem(s)].Data;
      with P^ do
      begin
        TheAn:=FieldByName('TheAn').AsString;
        U_Name:=FieldByName('U_Name').AsString;
        Plu:=FieldByName('Plu').AsString;
        U_Def:=FieldByName('U_Def').AsString;
        Man:=FieldByName('Man').AsString;
      end;
      arb.Items[arb.GetTextItem(s)].Data:=P;
      next
    end;
    Close
  end;
  MainWin.Caption:='AlbertII Paraphraser - '+CleanName(FName);
  modified:=False;
end;

{-----------------------------------------------------------------------}
procedure TMainWin.OpenParaphrase1Click(Sender: TObject);
```

```
var
  s:string;
begin
  OpenDialog1.Filter:='Paraphrase files (*.Pdb)|*.Pdb|All files (*.*)|*.*';
  OpenDialog1.FileName:='*.Pdb';
  If OpenDialog1.Execute then with OpenDialog1 do
  begin
    Close1Click(Sender);
    s:=OpenDialog1.FileName;
    i:=Pos('.',s);
    if i>0 then FName:=Copy(s,1,i-1)
    else FName:=FileName;
    AddInfo(arb,FName);CleanDeskTop(False);CleanObject;NewFile:=False;
  end
end;

{---------------------------------------------------------------------------}
procedure UseDictionary;
begin
  With MainWin.Table2 do
  begin
    Close;TableName:=MainWin.DicName.Caption;
    open;First;MainWin.UName.Clear;
    MainWin.UName.Items.Add('');
    while not eof do
    begin
      MainWin.UName.Items.Add(FieldByName('Name').AsString);
      Next
    end;Close
  end
end;

{---------------------------------------------------------------------------}
procedure TMainWin.FormCreate(Sender: TObject);
begin
  CleanDeskTop(True); modified:=False; NewFile:=False;HtmPath:='file://c:\memoire\in_out\';
  KeepSem:=True; KeepSpec:=False; TextAsc:=True;UseDictionary;FName:='';
end;

{---------------------------------------------------------------------------}
procedure TMainWin.arbClick(Sender: TObject);
var
  P:PMyObject;
  i,j,k:integer;
begin
  with arb.Items[arb.SelectedItem] do
  begin
    ArbItem:=Data;GoP.Visible:=(ArbItem^.Typ in [9,14]);
    ActBox.Visible:=(ArbItem^.Typ in [9,14]);
    ObjectBox.Enabled:=(ArbItem^.Typ<>-1);
    if ArbItem^.Typ<>-1 then
    begin
      ObjectBox.Caption:=ArbItem^.Name+' ';
      if not (ArbItem^.Typ in [9,14]) then
      begin
        UName.Text:=ObjectName(ArbItem);TheAn.Text:=ArbItem^.TheAn;
        UDef.Text:=ArbItem^.U_Def;Plu.Enabled:=(ArbItem^.IsPlu);
        LPlu.Enabled:=(ArbItem^.IsPlu); Plu.Text:=ArbItem^.Plu;
      end
      else
        Memo1.Text:=ArbItem^.U_Def;
      Auto.Checked:=ArbItem^.Auto; Manual.Checked:=ArbItem^.Manual;
      SDef.Text:=ArbItem^.Body; Ex.Text:=Generate(ArbItem);
```

```
    Man.Text:=ArbItem^.Man;
  end
  else
  begin
    CleanObject; ObjectBox.Caption:='Selected Object '; ObjectBox.Enabled:=False;
  end
 end
end;

{------------------------------------------------------------------------}
procedure TMainWin.NewBtClick(Sender: TObject);
begin
 New1Click(Sender)
end;

{------------------------------------------------------------------------}
procedure TMainWin.OpenBtClick(Sender: TObject);
begin
 OpenParaphrase1Click(Sender)
end;

{------------------------------------------------------------------------}
procedure TMainWin.SaveBtClick(Sender: TObject);
begin
 Save1Click(Sender)
end;

{------------------------------------------------------------------------}
procedure TMainWin.GenerateText1Click(Sender: TObject);
var
 P:PMyObject;
begin
 OutText.ShowModal;
 if OkGen then
 begin
  Preview.OutText.Clear;
  if TextAsc then
  begin
   Preview.Show;
   GenerateText
  end
  else
  begin
   Messag.Show;
   Messag.Canvas.Brush.Color:=clBtnFace;
   Messag.Canvas.TextOut(72,40,'Generating HTML Files.       ');
   GenerateHtml;
   Messag.Close;
   MessageDlg('Main Html file is: '+LowerCase(FName), mtInformation,[mbOk], 0);
  end
 end;
end;

{------------------------------------------------------------------------}
procedure TMainWin.PluKeyUp(Sender: TObject; var Key: Word;
 Shift: TShiftState);
begin
 ArbItem^.Plu:=Plu.Text; modified:=True;
end;

{------------------------------------------------------------------------}
procedure TMainWin.ManKeyUp(Sender: TObject; var Key: Word;
 Shift: TShiftState);
```

```
begin
 ArbItem:=arb.Items[arb.SelectedItem].Data;
 ArbItem^.Man:=Man.Text; modified:=True;
end;

{---------------------------------------------------------------------------}
function NewTable(TabName:String;Typ:integer):TTable;
var
 Table1:TTable;
 F:TextFile;
begin
 Table1:=TTable.Create(MainWin);
 with Table1 do
 begin
  Active := False;
  DatabaseName := TabName;
  TableName := TabName;
  TableType := ttParadox;
  with FieldDefs do
  begin
   Clear;
   if Typ=1 then
   begin
    Add('Text', ftString, 30,True);
    Add('TheAn', ftString, 3,False);
    Add('U_Name', ftString, 30,False);
    Add('Plu', ftString, 30,False);
    Add('U_Def', ftString, 150,False);
    Add('Man', ftString, 150,False);
    AssignFile(F, TabName+'.Pdb');Rewrite(F);CloseFile(F);
   end
   else
   begin
    Add('Name', ftString, 30,True);
    Add('Plural', ftString, 30,False);
    Add('Denotation', ftString, 100,False);
    with IndexDefs do
    begin
     Clear;
     Add(TabName, 'Name', [ixPrimary, ixUnique]);
    end;
   end
  end;
  CreateTable;AssignFile(F, TabName+'.dic');Rewrite(F);CloseFile(F);
 end;
 Newtable:=Table1
end;

{---------------------------------------------------------------------------}
procedure SaveSpec(arb:TOutline; FileName:TFileName);
var
 t:Word;
begin
 with MainWin.Table1 do
 begin
  if not FileExists(FileName+'.Pdb') then
   MainWin.Table1:=NewTable(FileName,1);
  TableName:=FileName;
  EmptyTable;
  Open;
  Append;
  FieldByName('Man').AsString:=Path;
  FieldByName('Text').AsString:='Path';
```

```
    Post;
    for i:=1 to arb.ItemCount do
    with arb do
    begin
      P:=Items[i].Data;
      s:=Items[i].Parent.Text;
      if Length(s)>2 then
        s:=Copy(s,1,2)+'_'
      else
        s:='';
      if P^.Typ<>-1 then
      begin
        Append;
        FieldByName('Text').AsString :=s+Items[i].Text;
        FieldByName('TheAn').AsString :=P^.TheAn;
        FieldByName('U_Name').AsString :=P^.U_Name;
        FieldByName('Plu').AsString :=P^.Plu;
        FieldByName('U_Def').AsString :=P^.U_Def;
        FieldByName('Man').AsString :=P^.Man;
        Post
      end
    end;
    Close
  end;
  with MainWin.Table2 do
  begin
    Close;TableName:=MainWin.DicName.Caption;
    Open;t:=mrNo;i:=1;
    while i<=arb.ItemCount do
    with arb do
    begin
      P:=Items[i].Data;
      if (P^.Typ<>-1) and (P^.U_Def<>'')then
      begin
        SetKey;
        FieldByName('Name').AsString :=P^.U_Name;
        if not GotoKey then
        begin
          if t=mrNo then
            t:=MessageDlg(Uppercase(P^.U_Name)+' is a new object. Update dictionary?',
                      mtInformation, [mbYes, mbAll, mbNo], 0);
          if t in [mrYes,mrAll] then
          begin
            Append;
            FieldByName('Name').AsString :=P^.U_Name;
            FieldByName('Plural').AsString :=Plural(p^);
            FieldByName('Denotation').AsString :=P^.U_Def;
            Post;
            if t=mrYes then t:=mrNo
          end
        end
      end; i:=i+1
    end;
    Close
  end
end;

{--------------------------------------------------------------------------}
procedure TMainWin.Save1Click(Sender: TObject);
begin
  if NewFile then Saveas1Click(Sender)
  else
  begin
```

```
    SaveSpec(arb,FName); modified:=False;
  end
end;

{------------------------------------------------------------------------}
procedure TMainWin.Close1Click(Sender: TObject);
var
  w:word;
begin
  CloseOk:=True;
  if modified then
  begin
    w:=MessageDlg('Save changes to file '+CleanName(FName),mtInformation, [mbYes, mbNo,mbCancel], 0);
    case w of
    mrYes:begin Save1Click(Sender); CleanDeskTop(True) end;
    mrNo :CleanDeskTop(True);
    mrCancel:CloseOk:=False;
    end
  end
  else
    CleanDeskTop(True);
end;

{------------------------------------------------------------------------}
procedure TMainWin.Saveas1Click(Sender: TObject);
begin
    SaveDialog1.Filter:='Paraphrase files (*.Pdb)|*.Pdb|All files (*.*)|*.*';
    s:=FName;
    SaveDialog1.FileName:=CleanName(FName)+'.Pdb';
    If SaveDialog1.Execute then with SaveDialog1 do
    begin
      s:=SaveDialog1.FileName;
      i:=Pos('.',s);
      if i>0 then
        FName:=Copy(s,1,i-1)
      else
        FName:=FileName;
      if FileExists(FileName) then
      begin
        if MessageDlg('File already exists, overwrite it?',mtInformation, [mbYes, mbNo], 0)=mrYes then
        begin
          SaveSpec(arb,FName);NewFile:=False;modified:=False;
        end
      end
      else
      begin
        SaveSpec(arb,FName);NewFile:=False;modified:=False;
      end;
      MainWin.Caption:='AlbertII Paraphraser - '+CleanName(FName);
    end
end;

{------------------------------------------------------------------------}
procedure TMainWin.PluEnter(Sender: TObject);
begin
  Plu.Text:=Plural(ArbItem^); Plu.SelectAll;
end;

{------------------------------------------------------------------------}
procedure TMainWin.Exit1Click(Sender: TObject);
begin
  Close1Click(Sender);
  if CloseOk then Close
```

```
end;

{----------------------------------------------------------------------------}
procedure TMainWin.Import1Click(Sender: TObject);
begin
 ImportWin.ShowModal;
 If OkGen and FileExists(ImportName+'.Pdb') then with Table1 do
 begin
  TableName:=ImportName;Close;
  Open;First;
  if ImpBasic then
  begin
   While Copy(FieldByName('Text').AsString,1,3)<>'BA_' do
    Next;
   s:=FieldByName('Text').AsString;
   While Copy(s,1,3)='BA_' do
   begin
    s:=Copy(s,4,Length(s)-3);
    i:=arb.GetTextItem(s);
    if i>0 then
    begin
     P:=arb.Items[i].Data;
     P^.U_Name:=FieldByName('U_Name').AsString;
     P^.U_Def:=FieldByName('U_Def').AsString;
     P^.Plu:=FieldByName('Plu').AsString;
     P^.Man:=FieldByName('Man').AsString;
    end;
    Next; s:=FieldByName('Text').AsString;
   end;
  end;
  if ImpCons then
  begin
   While Copy(FieldByName('Text').AsString,1,3)<>'CO_' do Next;
   s:=FieldByName('Text').AsString;
   While Copy(s,1,3)='CO_' do
   begin
    s:=Copy(s,4,Length(s)-3); i:=arb.GetTextItem(s);
    if i>0 then
    begin
     P:=arb.Items[i].Data;
     P^.U_Name:=FieldByName('U_Name').AsString;
     P^.U_Def:=FieldByName('U_Def').AsString;
     P^.Plu:=FieldByName('Plu').AsString;
     P^.Man:=FieldByName('Man').AsString;
    end;
    Next; s:=FieldByName('Text').AsString;
   end;
  end;
  if ImpOp then
  begin
   While Copy(FieldByName('Text').AsString,1,3)<>'OP_' do
    Next;
   s:=FieldByName('Text').AsString;
   While Copy(s,1,3)='OP_' do
   begin
    s:=Copy(s,4,Length(s)-3); i:=arb.GetTextItem(s);
    if i>0 then
    begin
     P:=arb.Items[i].Data;
     P^.U_Def:=FieldByName('U_Def').AsString;
     P^.Man:=FieldByName('Man').AsString;
    end;
    Next; s:=FieldByName('Text').AsString;
```

```
      end;
    end;
    Close;
  end
  else if OkGen then
    MessageDlg('File not found: '+CleanName(ImportName)+'.db .', mtInformation,[mbOk], 0);
  modified:=True
end;

{--------------------------------------------------------------------------}
procedure TMainWin.ImpBtClick(Sender: TObject);
begin
  Import1Click(Sender)
end;

{--------------------------------------------------------------------------}
procedure TMainWin.UDefKeyUp(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  ArbItem^.U_Def:=UDef.Text; modified:=True; Ex.Text:=Generate(ArbItem)
end;

{--------------------------------------------------------------------------}
procedure TMainWin.ATMClick(Sender: TObject);
begin
  Man.Lines.Insert(0,Ex.Text); ActiveControl:=Man; ATM.Enabled:=False
end;

{--------------------------------------------------------------------------}
procedure TMainWin.TheAnChange(Sender: TObject);
begin
  ArbItem^.TheAn:=TheAn.Items[TheAn.ItemIndex];
  TheAn.Text:=ArbItem^.TheAn; Ex.Text:=Generate(ArbItem);
end;

{--------------------------------------------------------------------------}
procedure TMainWin.GoPClick(Sender: TObject);
begin
  Ex.text:=Generate(ArbItem); GoP.Enabled:=False; Modified:=True
end;

{--------------------------------------------------------------------------}
procedure TMainWin.Memo1Change(Sender: TObject);
begin
  GoP.Enabled:=(Memo1.Text<>''); ArbItem^.U_Def:=Memo1.Text
end;

{--------------------------------------------------------------------------}
procedure TMainWin.TypesDblClick(Sender: TObject);
begin
  if memo1.SelText<>'' then
    memo1.SelText:=Types.Items[Types.ItemIndex]+' '
  else
    memo1.SelText:=Types.Items[Types.ItemIndex];
  ActiveControl:=memo1
end;

{--------------------------------------------------------------------------}
procedure TMainWin.ExChange(Sender: TObject);
begin
  ATM.Enabled:=(Ex.Text<>'')
end;
```

```
{-----------------------------------------------------------------------------}
procedure TMainWin.Use1Click(Sender: TObject);
var
  s:string;
begin
  OpenDialog1.Filter:='Dictionaries (*.dic)|*.dic|All files (*.*)|*.*';
  OpenDialog1.FileName:='.dic';
  If OpenDialog1.Execute then with OpenDialog1 do
  begin
    s:=OpenDialog1.FileName; i:=Pos('.',s);
    if i>0 then s:=Copy(s,1,i-1);
    DicName.Caption:=s; Query1.Sql.Strings[1]:='From In_Out\'+CleanName(s);
    if not FileExists(s+'.db') then Table2:=NewTable(s,2);s:='';
    if UName.Text<>'' then s:=UName.Text;UseDictionary;UName.Text:=s
  end
end;

{-----------------------------------------------------------------------------}
procedure TMainWin.UNameChange(Sender: TObject);
begin
  ArbItem^.U_Name:=UName.Text;
  ArbItem^.TheAn:=ObjectTheAn(UName.Text);
  if (UName.ItemIndex>-1) and (UName.Text=UName.Items[UName.ItemIndex]) then with Query1 do
  begin
    Close;Params[0].AsString:=UName.Text;
    if FieldCount>0 then Fields[0].Clear;
    if FieldCount>0 then Fields[1].Clear;Open;
    Plu.Text:=Fields[0].Text;ArbItem^.Plu:=Plu.Text;
    UDef.Text:=Fields[1].Text;ArbItem^.U_Def:=UDef.Text;
  end;
  modified:=True; Ex.text:=Generate(ArbItem)
end;

{-----------------------------------------------------------------------------}
procedure TMainWin.AutoClick(Sender: TObject);
begin
  ArbItem^.Auto:=Auto.Checked; SemOk.Enabled:=Auto.Checked; Semko.Enabled:=Auto.Checked;
end;

{-----------------------------------------------------------------------------}
procedure TMainWin.ManualClick(Sender: TObject);
begin
  ArbItem^.Manual:=Manual.Checked
end;

{-----------------------------------------------------------------------------}
procedure TMainWin.SemOkClick(Sender: TObject);
begin
  KeepSem:=SemOk.Checked; Ex.Text:=Generate(ArbItem)
end;

{-----------------------------------------------------------------------------}
procedure TMainWin.SemKoClick(Sender: TObject);
begin
  KeepSem:=SemOk.Checked; Ex.Text:=Generate(ArbItem)
end;

{-----------------------------------------------------------------------------}
procedure TMainWin.ManChange(Sender: TObject);
begin
  ArbItem^.Man:=Man.Text;
end;
end.
```

```
unit Parser;

interface

uses Classes, SysUtils, WinTypes, WinProcs, SSLexU, SSExcept, SSYaccU,
   Globs;

const
 ALexExpressionListMain         = 0;
 ALexExpressionListIgnoredPart  = 1;

const
 ALexOneLine          = 2;
 ALexSpecStart        = 14;
 ALexBasicTypes       = 15;
 ALexConsTypes        = 16;
 ALexOps              = 17;
 ALexSociety          = 18;
 ALexAgent            = 19;
 ALexStateComp        = 20;
 ALexActions          = 21;
 ALexNumber           = 22;
 ALexIdentifier       = 23;
 ALexBag              = 24;
 ALexCP               = 25;
 ALexEnum             = 26;
 ALexSeq              = 27;
 ALexSet              = 28;
 ALexTable            = 29;
 ALexUnion            = 30;
 ALexInstanceComp     = 31;
 ALexSetComp          = 32;
 ALexTableComp        = 33;
 ALexSeqComp          = 34;
 ALexIndex            = 35;
 ALexDerived          = 36;
 ALexCartProd         = 37;
 ALexEqual            = 38;
 ALexStar             = 39;
 ALexLeftBrace        = 40;
 ALexRightBrace       = 41;
 ALexLeftBracket      = 42;
 ALexRightBracket     = 43;
 ALexColumn           = 44;
 ALexComma            = 45;
 ALexLeftParen        = 46;
 ALexRightParen       = 47;
 ALexDot              = 48;
 ALexRightArrow       = 49;

const
 AYaccStart           = 1;
 AYaccOneLineComm     = 2;
 AYaccMultiLineCom    = 3;
 AYaccTypesOpsDec     = 4;
 AYaccBasicVide       = 5;
 AYaccBasicDec        = 6;
 AYaccBasicTypeNo     = 7;
 AYaccBasicType       = 8;
 AYaccBType           = 9;
 AYaccBTypeList       = 10;
 AYaccConstVide       = 11;
 AYaccConstDec        = 12;
```

```
AYaccCType = 13;
AYaccCTypeList = 14;
AYaccConstTypeNo = 15;
AYaccConstType = 16;
AYaccUdefExpr = 17;
AYaccCpExpr = 18;
AYaccSetExpr = 19;
AYaccSeqExpr = 20;
AYaccBagExpr = 21;
AYaccTabExpr = 22;
AYaccUnExpr = 23;
AYaccEnumExpr = 24;
AYaccTypeSing = 25;
AYaccTypeList = 26;
AYaccSelector = 27;
AYaccSelectorList = 28;
AYaccOpsVide = 29;
AYaccOpsDecls = 30;
AYaccOneOp = 31;
AYaccOpsList = 32;
AYaccOpDeclNo = 33;
AYaccOpDecl = 34;
AYaccOpArityNoDom = 35;
AYaccOpArityDom = 36;
AYaccOpDom = 37;
AYaccOpDoms = 38;
AYaccRootSocDec = 39;
AYaccOneSociety = 40;
AYaccSocietyList = 41;
AYaccSociety = 42;
AYaccSubSocAg = 43;
AYaccSubSocAgList = 44;
AYaccSubSocAgSing = 45;
AYaccSubSocAgMult = 46;
AYaccOneAgent = 47;
AYaccAgentList = 48;
AYaccAgent = 49;
AYaccRootSocName = 50;
AYaccSubSocAgentName = 51;
AYaccNoStateComp = 52;
AYaccStateCompsList = 53;
AYaccCompListEnd = 54;
AYaccCompList = 55;
AYaccFixedCompNo = 56;
AYaccVarCompNo = 57;
AYaccFixedComp = 58;
AYaccVarComp = 59;
AYaccNoDerivation = 60;
AYaccDerivation = 61;
AYaccOneCompName = 62;
AYaccManyCompNames = 63;
AYaccNoExportation = 64;
AYaccExportation = 65;
AYaccSetSeqInstComp = 66;
AYaccTableComp = 67;
AYaccSetComp = 68;
AYaccSeqComp = 69;
AYaccInstanceComp = 70;
AYaccActionNul = 71;
AYaccActionDecl = 72;
AYaccOneAction = 73;
AYaccActionList = 74;
AYaccStarActionDecNo = 75;
```

```
  AYaccActionDecNo            = 76;
  AYaccStarActionDec          = 77;
  AYaccActionDec              = 78;
  AYaccActionNoParams         = 79;
  AYaccActionParams           = 80;

type

  AYaccStackElement = class(SSYaccStackElement)
  public
  Value  : MyObject;

  constructor Create;
end;

  AYaccClass = class(SSYacc)
  public
  FinalValue : MyObject;

  function StackElement : SSYaccStackElement; override;
  function Reduce( TheProduction, TheSize : Longint) : SSYaccStackElement; override;
end;

function AnalyseSpec (FileName:PChar) : String;

implementation

function AnalyseSpec(FileName:PChar): String;
var
  Lexer      : SSLex;
  Parser     : AYaccClass;
  LexTable   : SSLexTable;
  ParseTable : SSYaccTable;
  Consumer   : SSLexFileConsumer;
begin
  Consumer := SSLexFileConsumer.Create(FileName,32768,0,SSLexTextMode);
  LexTable := SSLexTable.Create( 'c:\memoire\albert.dfa');
  Lexer := SSLex.Create( Consumer, LexTable);
  ParseTable := SSYaccTable.Create( 'c:\memoire\albert.llr');
  Parser := AYaccClass.CreateLex( Lexer, ParseTable);
  MySpec.BTypes:=TList.Create;
  MySpec.CTypes:=TList.Create;
  MySpec.Operations:=TList.Create;
  MySpec.Societies:=TList.Create;
  Parser.Parse;
  Result := Parser.FinalValue.name;
end;

constructor AYaccStackElement.Create;
begin
  inherited Create;
  with Value do
  begin
    Name :=''; U_Name :=''; U_Def :=''; Body:=''; Plu:=''; Man:=''; IsPlu:=False;
  end
end;

function AYaccClass.StackElement : SSYaccStackElement;
begin
  Result := AYaccStackElement.Create;
end;

function AYaccClass.Reduce( TheProduction, TheSize:Longint):SSYaccStackElement;
```

```
var
  i,j,k : integer;
  E    : AYaccStackElement;
  E0   : AYaccStackElement;
  E1   : AYaccStackElement;
  E2   : AYaccStackElement;
  E3   : AYaccStackElement;
  E4   : AYaccStackElement;

begin
  Result:=nil;
  case TheProduction of
   AYaccStart:
   { spec -> SpecStart Identifier TypesOpsDec RootSoc }
   begin
     E1:= AYaccStackElement(ElementFromProduction(1));
     FinalValue.Name :=StrPas(E1.Lexeme.Buffer);
   end
   ;

    AYaccOneLineComm:
   { Comment -> OneLine }
   begin
     E:= AYaccStackElement(StackElement);
     E0:= AYaccStackElement(ElementFromProduction(0));
     E.Value.Man:=Copy(StrPas(E0.Lexeme.Buffer),3,Length(StrPas(E0.Lexeme.Buffer))-3);
     Result:=E
   end
   ;

   AYaccMultiLineCom:
   { Comment -> OneLine Comment }
   begin
     E:= AYaccStackElement(StackElement);
     E0:= AYaccStackElement(ElementFromProduction(0));
     E1:= AYaccStackElement(ElementFromProduction(1));
     E.Value:=E1.Value;
     E.Value.Man:=Copy(StrPas(E0.Lexeme.Buffer),3,Length(StrPas(E0.Lexeme.Buffer))-3)+' '+E.Value.Man;
     Result:=E
   end
   ;

  AYaccTypesOpsDec:
   { TypesOpsDec -> BasicDec ConstDec OpsDec }
   ;

   AYaccBasicVide:
   { BasicDec -> BasicTypes }
   ;

   AYaccBasicDec:
   { BasicDec -> BasicTypes BTypeList }
   ;

   AYaccBasicTypeNo:
   { BasicType -> Identifier }
   begin
     E0:= AYaccStackElement(ElementFromProduction(0));
     New(PE);
     PE^.Name:=StrPas(E0.Lexeme.Buffer);
     PE^.Body:='Basic type';
     PE^.Typ:=0;
     MySpec.BTypes.Add(PE);
```

```
end
;

AYaccBasicType:
{ BasicType -> Comment Identifier }
begin
 E0:= AYaccStackElement(ElementFromProduction(0));
 E1:= AYaccStackElement(ElementFromProduction(1));
 New(PE);
 PE^.Name:=StrPas(E1.Lexeme.Buffer);
 PE^.Body:='Basic type';
 PE^.Typ:=0;
 PE^.Man:=E0.Value.Man;
 MySpec.BTypes.Add(PE);
end
;

AYaccBType:
{ BTypeList -> BasicType }
;

AYaccBTypeList:
{ BTypeList -> BasicType BTypeList }
;

AYaccConstVide:
{ ConstDec -> ConsTypes }
;

AYaccConstDec:
{ ConstDec -> ConsTypes CTypeList }
;

AYaccCType:
{ CTypeList -> ConstType }
begin
 E0:= AYaccStackElement(ElementFromProduction(0));
 New(PE);
 PE^:=E0.Value;
 MySpec.CTypes.Add(PE);
end
;

AYaccCTypeList:
{ CTypeList -> ConstType CTypeList }
begin
 E0:= AYaccStackElement(ElementFromProduction(0));
 New(PE);
 PE^:=E0.Value;
 MySpec.CTypes.Insert(0,PE);
end
;

AYaccConstTypeNo:
{ ConstType -> Identifier Equal TypeExpr }
 begin
 E:= AYaccStackElement(StackElement);
 E0:= AYaccStackElement(ElementFromProduction(0));
 E2:= AYaccStackElement(ElementFromProduction(2));
 E.Value:=E2.Value;
 E.Value.Name:=StrPas(E0.Lexeme.Buffer);
 E.Value.Body:=StrPas(E0.Lexeme.Buffer)+' = '+E2.Value.Body;
 Result:=E;
```

```
end
;

AYaccConstType:
{ ConstType -> Comment Identifier Equal TypeExpr }
 begin
  E:= AYaccStackElement(StackElement);
  E0:= AYaccStackElement(ElementFromProduction(0));
  E1:= AYaccStackElement(ElementFromProduction(1));
  E3:= AYaccStackElement(ElementFromProduction(3));
  E.Value:=E3.Value;
  E.Value.Man:=E0.Value.Man;
  E.Value.Name:=StrPas(E1.Lexeme.Buffer);
  E.Value.Body:=StrPas(E1.Lexeme.Buffer)+' = '+E3.Value.Body;
  Result:=E;
end
;

AYaccUdefExpr:
{ TypeExpr -> Identifier }
begin
  E:= AYaccStackElement(StackElement);
  E0:= AYaccStackElement(ElementFromProduction(0));
  E.Value.Name:=StrPas(E0.Lexeme.Buffer);
  E.Value.Typ:=1;
  E.Value.Body:=E.Value.Name;
  E.Value.Rd:=E.Value.Name;
  Result:=E
end
;

AYaccCpExpr:
{ TypeExpr -> CP LeftBracket Selector RightBracket }
begin
  E:= AYaccStackElement(StackElement);
  E2:= AYaccStackElement(ElementFromProduction(2));
  E.Value:=E2.Value;
  E.Value.Name:='CP';
  E.Value.Typ:=5;
  E.Value.Body:= 'CP[ '+E2.Value.Body+' ]';
  Result:=E;
end
;

AYaccSetExpr:
{ TypeExpr -> Set LeftBracket TypeExpr RightBracket }
begin
  E:= AYaccStackElement(StackElement);
  E2:= AYaccStackElement(ElementFromProduction(2));
  E.Value.Value:=TList.Create;
  New(PE);
  PE^:=E2.Value;
  E.Value.Name:='SET';
  E.Value.Body:= 'SET[ '+E2.Value.Body+' ]';
  E.Value.Typ:=2;
  E.Value.Value.Add(PE);
  Result:=E;
end
;

AYaccSeqExpr:
{ TypeExpr -> Seq LeftBracket TypeExpr RightBracket }
begin
```

```
  E:= AYaccStackElement(StackElement);
  E2:= AYaccStackElement(ElementFromProduction(2));
  E.Value.Value:=TList.Create;
  New(PE);
  PE^:=E2.Value;
  E.Value.Name:='SEQ';
  E.Value.Body:= 'SEQ[ '+E2.Value.Body+' ]';
  E.Value.Typ:=3;
  E.Value.Value.Add(PE);
  Result:=E;
end
;

AYaccBagExpr:
{ TypeExpr -> Bag LeftBracket TypeExpr RightBracket }
begin
  E:= AYaccStackElement(StackElement);
  E2:= AYaccStackElement(ElementFromProduction(2));
  E.Value.Value:=TList.Create;
  New(PE);
  PE^:=E2.Value;
  E.Value.Name:='BAG';
  E.Value.Body:= 'BAG[ '+E2.Value.Body+' ]';
  E.Value.Typ:=4;
  E.Value.Value.Add(PE);
  Result:=E;
end
;

AYaccTabExpr:
{ TypeExpr -> Table LeftBracket TypeExpr RightArrow TypeExpr RightBracket }
begin
  E:= AYaccStackElement(StackElement);
  E2:= AYaccStackElement(ElementFromProduction(2));
  E4:= AYaccStackElement(ElementFromProduction(4));
  New(PE);
  PE^:=E2.Value;
  E.Value.Dom:=TList.Create;
  E.Value.Codom:=TList.Create;
  E.Value.Dom.Add(PE);
  New(PE);
  PE^:=E4.Value;
  E.Value.Codom.Add(PE);
  E.Value.Body:= 'TABLE[ '+E2.Value.Body+' --> '+E4.Value.Body+' ]';
  E.Value.Typ:=8;
  Result:=E;
end
;

AYaccUnExpr:
{ TypeExpr -> Union LeftBracket TypeList RightBracket }
begin
  E:= AYaccStackElement(StackElement);
  E2:= AYaccStackElement(ElementFromProduction(2));
  E.Value:=E2.Value;
  E.Value.Name:='UNION';
  E.Value.Body:= 'UNION[ '+E2.Value.Body+' ]';
  E.Value.Typ:=6;
  Result:=E;
end
;

AYaccEnumExpr:
```

```
{ TypeExpr -> Enum LeftBracket TypeList RightBracket }
begin
  E:= AYaccStackElement(StackElement);
  E2:= AYaccStackElement(ElementFromProduction(2));
  E.Value:=E2.Value;
  E.Value.Name:='ENUM';
  E.Value.Body:= 'ENUM[ '+E2.Value.Body+' ]';
  E.Value.Typ:=7;
  Result:=E;
end
;

AYaccTypeSing:
{ TypeList -> TypeExpr }
begin
  E:= AYaccStackElement(StackElement);
  E0:= AYaccStackElement(ElementFromProduction(0));
  E.Value.Value:= TList.Create;
  New(PE);
  PE^:= E0.Value;
  E.Value.Body:= E0.Value.Body;
  E.Value.Value.Add(PE);
  Result:=E;
end
;

AYaccTypeList:
{ TypeList -> TypeExpr Comma TypeList }
begin
  E:= AYaccStackElement(StackElement);
  E0:= AYaccStackElement(ElementFromProduction(0));
  E2:= AYaccStackElement(ElementFromProduction(2));
  E.Value:=E2.Value;
  New(PE);
  PE^:= E0.Value;
  E.Value.Body:= E0.Value.Body+' , '+E.Value.Body;
  E.Value.Value.Insert(0,PE);
  Result:=E;
end
;

AYaccSelector:
{ Selector -> Identifier Column TypeExpr }
begin
  E:= AYaccStackElement(StackElement);
  E0:= AYaccStackElement(ElementFromProduction(0));
  E2:= AYaccStackElement(ElementFromProduction(2));
  E.Value.Value:= TList.Create;
  New(PE);
  PE^:= E2.Value;
  if PE^.Typ=1 then
    PE^.Typ:=15;
  PE^.Name:=StrPas(E0.Lexeme.Buffer);
  E.Value.Body:= PE^.Name+' : '+E2.Value.Body;;
  E.Value.Value.Add(PE);
  Result:=E;
end
;

AYaccSelectorList:
{ Selector -> Identifier Column TypeExpr Comma Selector }
begin
  E := AYaccStackElement(StackElement);
```

```
  E0:= AYaccStackElement(ElementFromProduction(0));
  E2:= AYaccStackElement(ElementFromProduction(2));
  E4:= AYaccStackElement(ElementFromProduction(4));
  E.Value:=E4.Value;
  New(PE);
  PE^:=E2.Value;
  if PE^.Typ=1 then
    PE^.Typ:=15;
  PE^.Name:=StrPas(E0.Lexeme.Buffer);
  E.Value.Body:= PE^.Name+' : '+E2.Value.Body+' , '+E4.Value.Body;
  E.Value.Value.Insert(0,PE);
  Result:=E;
end
;

AYaccOpsVide:
{ OpsDec -> Ops }
begin
  MySpec.Operations:=nil
end
;

AYaccOpsDecls:
{ OpsDec -> Ops OpsList }
;

AYaccOneOp:
{ OpsList -> OpDec }
begin
  E0:= AYaccStackElement(ElementFromProduction(0));
  New(PE);
  PE^:=E0.Value;
  MySpec.Operations.Add(PE);
end
;

AYaccOpsList:
{ OpsList -> OpDec OpsList }
begin
  E0:= AYaccStackElement(ElementFromProduction(0));
  New(PE);
  PE^:=E0.Value;
  MySpec.Operations.Insert(0,PE);
end
;

AYaccOpDeclNo:
{ OpDec -> Identifier Column OpArity }
begin
  E:= AYaccStackElement(StackElement);
  E2:= AYaccStackElement(ElementFromProduction(2));
  E0:= AYaccStackElement(ElementFromProduction(0));
  E.Value:=E2.Value;
  E.Value.Typ:=9;
  E.Value.Name:=StrPas(E0.Lexeme.Buffer);
  E.Value.Body:=E.Value.Name+' : '+E2.Value.Body;
  Result:=E;
end
;

AYaccOpDecl:
{ OpDec -> Comment Identifier Column OpArity }
begin
```

```
  E:= AYaccStackElement(StackElement);
  E0:= AYaccStackElement(ElementFromProduction(0));
  E1:= AYaccStackElement(ElementFromProduction(1));
  E3:= AYaccStackElement(ElementFromProduction(3));
  E.Value:=E3.Value;
  E.Value.Man:=E0.Value.Man;
  E.Value.Name:=StrPas(E1.Lexeme.Buffer);
  E.Value.Body:=StrPas(E1.Lexeme.Buffer)+' = '+E3.Value.Body;
  E.Value.Typ:=9;
  Result:=E;
end
;

AYaccOpArityNoDom:
{ OpArity -> RightArrow TypeExpr }
begin
  E:= AYaccStackElement(StackElement);
  E1:= AYaccStackElement(ElementFromProduction(1));
  E.Value.Dom:=nil;
  New(PE);
  PE^:=E1.Value;
  E.Value.Codom.Add(PE);
  E.Value.Body:=' --> '+E1.Value.Body;
  Result:=E;
end
;

AYaccOpArityDom:
{ OpArity -> OpDom RightArrow TypeExpr }
begin
  E:= AYaccStackElement(StackElement);
  E0:= AYaccStackElement(ElementFromProduction(0));
  E2:= AYaccStackElement(ElementFromProduction(2));
  E.Value.Dom:=TList.Create;
  E.Value.Codom:=TList.Create;
  E.Value.Dom:=E0.Value.Dom;
  New(PE);
  PE^:=E2.Value;
  E.Value.Codom.Add(PE);
  E.Value.Body:=E0.Value.Body+' --> '+E2.Value.Body;
  Result:=E;
end
;

AYaccOpDom:
{ OpDom -> TypeExpr }
begin
  E:= AYaccStackElement(StackElement);
  E0:= AYaccStackElement(ElementFromProduction(0));
  New(PE);
  PE^:= E0.Value;
  E.Value.Dom:=TList.Create;
  E.Value.Dom.Add(PE);
  E.Value.Body:=E0.Value.Body;
  Result:=E;
end
;

AYaccOpDoms:
{ OpDom -> TypeExpr CartProd OpDom }
begin
  E:= AYaccStackElement(StackElement);
  E0:= AYaccStackElement(ElementFromProduction(0));
```

```
  E2:= AYaccStackElement(ElementFromProduction(2));
  E.Value:=E2.Value;
  New(PE);
  PE^:= E0.Value;
  E.Value.Dom.Insert(0,PE);
  E.Value.Body:=E0.Value.Body+' X '+E2.Value.Body;
  Result:=E;
end
;

AYaccRootSocDec:
{ RootSoc -> SocList }
;

AYaccOneSociety:
{ SocList -> SocDec AgentList }
begin
  E0:= AYaccStackElement(ElementFromProduction(0));
  E1:= AYaccStackElement(ElementFromProduction(1));
  New(PE);
  PE^:=E0.Value;
  PE^.Agents:=TList.Create;
  PE^.Agents:=E1.Value.Agents;
  MySpec.Societies.Add(PE);
end
;

AYaccSocietyList:
{ SocList -> SocDec AgentList SocList }
begin
  E0:= AYaccStackElement(ElementFromProduction(0));
  E1:= AYaccStackElement(ElementFromProduction(1));
  New(PE);
  PE^:=E0.Value;
  PE^.IsPlu:=False;
  PE^.Agents:=TList.Create;
  PE^.Agents:=E1.Value.Agents;
  MySpec.Societies.Insert(0,PE);
end
;

AYaccSociety:
{ SocDec -> Society SocAgentName SubSocAgList }
begin
  E:= AYaccStackElement(StackElement);
  E1:= AYaccStackElement(ElementFromProduction(1));
  E3:= AYaccStackElement(ElementFromProduction(2));
  E.Value.SocAgents:=E3.Value.SocAgents;
  E.Value.Name:=E1.Value.Name;
  E.Value.Typ:=10;
  E.Value.Body:=E1.Value.Name+' : '+E3.Value.Body;
  Result:=E;
end
;

AYaccSubSocAg:
{ SubSocAgList -> SubSocAg }
begin
  E:= AYaccStackElement(StackElement);
  E0:= AYaccStackElement(ElementFromProduction(0));
  E.Value.SocAgents:=TList.Create;
  New(PE);
  PE^:=E0.Value;
```

```
  E.Value.Body:=E0.Value.Body;
  E.Value.SocAgents.Add(PE);
  Result:=E
end
;

AYaccSubSocAgList:
{ SubSocAgList -> SubSocAg SubSocAgList }
begin
  E:= AYaccStackElement(StackElement);
  E0:= AYaccStackElement(ElementFromProduction(0));
  E2:= AYaccStackElement(ElementFromProduction(1));
  E.Value:=E2.Value;
  E.Value.Body:=E0.Value.Body+' , '+E.Value.Body;
  New(PE);
  PE^:=E0.Value;
  E.Value.SocAgents.Insert(0,PE);
  Result:=E
end
;

AYaccSubSocAgSing:
{ SubSocAg -> LeftParen Identifier RightParen }
begin
  E:= AYaccStackElement(StackElement);
  E1:= AYaccStackElement(ElementFromProduction(1));
  E.Value.Typ:=13;
  E.Value.IsPlu:=False;
  E.Value.Name:=StrPas(E1.Lexeme.Buffer);
  E.Value.Body:='('+E.Value.Name+')';
  Result:=E
end
;

AYaccSubSocAgMult:
{ SubSocAg -> LeftParen Identifier RightParen RightParen RightParen }
begin
  E:= AYaccStackElement(StackElement);
  E1:= AYaccStackElement(ElementFromProduction(1));
  E.Value.Name:=StrPas(E1.Lexeme.Buffer);
  E.Value.Typ:=13;
  E.Value.IsPlu:=True;
  E.Value.Body:='('+E.Value.Name+')))';
  Result:=E
end
;

AYaccOneAgent:
{ AgentList -> AgentDec }
begin
  E:= AYaccStackElement(StackElement);
  E0:= AYaccStackElement(ElementFromProduction(0));
  New(PE);
  PE^:=E0.Value;
  PE^.Typ:=11;
  E.Value.Agents:=TList.Create;
  E.Value.Agents.Add(PE);
  Result:=E
end
;

AYaccAgentList:
{ AgentList -> AgentDec AgentList }
```

```
begin
  E:= AYaccStackElement(StackElement);
  E0:= AYaccStackElement(ElementFromProduction(0));
  E1:= AYaccStackElement(ElementFromProduction(1));
  E.Value:=E1.Value;
  New(PE);
  PE^:=E0.Value;
  PE^.Typ:=11;
  E.Value.Agents.Insert(0,PE);
  Result:=E
end
;

AYaccAgent:
{ AgentDec -> Agent SocAgentName StateCompDec ActionsDec }
begin
  E:= AYaccStackElement(StackElement);
  E1:= AYaccStackElement(ElementFromProduction(1));
  E.Value:=E1.Value;
  with E.Value do
  begin
    StateComps:=TList.Create;
    Actions:=TList.Create;
    E0:= AYaccStackElement(ElementFromProduction(2));
    StateComps:=E0.Value.StateComps;
    E1:= AYaccStackElement(ElementFromProduction(3));
    Actions:=E1.Value.Actions;
  end;
  Result:=E
end
;

AYaccRootSocName:
{ SocAgentName -> Identifier }
begin
  E:= AYaccStackElement(StackElement);
  E0:= AYaccStackElement(ElementFromProduction(0));
  E.Value.Name:=StrPas(E0.Lexeme.Buffer);
  Result:=E
end
;

AYaccSubSocAgentName:
{ SocAgentName -> Identifier Dot SocAgentName }
begin
  E:= AYaccStackElement(StackElement);
  E0:= AYaccStackElement(ElementFromProduction(0));
  E2:= AYaccStackElement(ElementFromProduction(2));
  E.Value.Name:=StrPas(E0.Lexeme.Buffer)+'.'+E2.Value.Name;
  Result:=E
end
;

AYaccNoStateComp:
{ StateCompDec -> StateComp }
begin
  E:= AYaccStackElement(StackElement);
  E.Value.StateComps:=nil;
  Result:=E
end
;

AYaccStateCompsList:
```

```
{ StateCompDec -> StateComp CompList }
begin
  E:= AYaccStackElement(ElementFromProduction(1));
  Result:=E
end
;

AYaccCompListEnd:
{ CompList -> CompDec }
begin
  E:= AYaccStackElement(StackElement);
  E0:= AYaccStackElement(ElementFromProduction(0));
  E.Value.StateComps:=TList.Create;
  New(PE);
  PE^:=E0.Value;
  PE^.Typ:=12;
  PE^.IsPlu:=False;
  E.Value.StateComps.Add(PE);
  Result:=E
end
;

AYaccCompList:
{ CompList -> CompDec CompList }
begin
  E:= AYaccStackElement(StackElement);
  E0:= AYaccStackElement(ElementFromProduction(0));
  E2:= AYaccStackElement(ElementFromProduction(1));
  E.Value:=E2.Value;
  New(PE);
  PE^:=E0.Value;
  PE^.Typ:=12;
  PE^.IsPlu:=False;
  E.Value.StateComps.Insert(0,PE);
  Result:=E
end
;

AYaccFixedCompNo:
{ CompDec -> Star Comp Derivs Exports }
begin
  E:= AYaccStackElement(StackElement);
  E1:= AYaccStackElement(ElementFromProduction(1));
  E2:= AYaccStackElement(ElementFromProduction(2));
  E3:= AYaccStackElement(ElementFromProduction(3));
  New(PE);
  E.Value:=E1.Value;
  E.Value.Fix:=True;
  E.Value.Deriv:=TStringList.Create;
  E.Value.Exprt:=TStringList.Create;
  E.Value.Deriv:=E2.Value.Deriv;
  E.Value.Exprt:=E3.Value.Exprt;
  E.Value.Body:='*'+E1.Value.Body+E2.Value.Body+E3.Value.Body;
  Result:=E
end
;

AYaccVarCompNo:
{ CompDec -> Comp Derivs Exports }
begin
  E:= AYaccStackElement(StackElement);
  E0:= AYaccStackElement(ElementFromProduction(0));
  E1:= AYaccStackElement(ElementFromProduction(1));
```

```
  E2:= AYaccStackElement(ElementFromProduction(2));
  New(PE);
  E.Value:=E0.Value;
  E.Value.Fix:=False;
  E.Value.Exprt:=TStringList.Create;
  E.Value.Deriv:=TStringList.Create;
  E.Value.Deriv:=E1.Value.Deriv;
  E.Value.Exprt:=E2.Value.Exprt;
  E.Value.Body:=E0.Value.Body+E1.Value.Body+E2.Value.Body;
  Result:=E
end
;

AYaccFixedComp:
{ CompDec -> Comment Star Comp Derivs Exports }
begin
  E:= AYaccStackElement(StackElement);
  E0:= AYaccStackElement(ElementFromProduction(0));
  E2:= AYaccStackElement(ElementFromProduction(2));
  E3:= AYaccStackElement(ElementFromProduction(3));
  E4:= AYaccStackElement(ElementFromProduction(4));
  New(PE);
  E.Value:=E2.Value;
  E.Value.Fix:=True;
  E.Value.Deriv:=TStringList.Create;
  E.Value.Exprt:=TStringList.Create;
  E.Value.Deriv:=E3.Value.Deriv;
  E.Value.Exprt:=E4.Value.Exprt;
  E.Value.Body:='*'+E2.Value.Body+E3.Value.Body+E4.Value.Body;
  E.Value.Man:=E0.Value.Man;
  Result:=E
end
;

AYaccVarComp:
{ CompDec -> Comment Comp Derivs Exports }
begin
  E:= AYaccStackElement(StackElement);
  E0:= AYaccStackElement(ElementFromProduction(0));
  E1:= AYaccStackElement(ElementFromProduction(1));
  E2:= AYaccStackElement(ElementFromProduction(2));
  E3:= AYaccStackElement(ElementFromProduction(3));
  New(PE);
  E.Value:=E1.Value;
  E.Value.Fix:=False;
  E.Value.Exprt:=TStringList.Create;
  E.Value.Deriv:=TStringList.Create;
  E.Value.Deriv:=E2.Value.Deriv;
  E.Value.Exprt:=E3.Value.Exprt;
  E.Value.Body:=E1.Value.Body+E2.Value.Body+E3.Value.Body;
  E.Value.Man:=E0.Value.Man;
  Result:=E
end
;

AYaccNoDerivation:
{ Derivs -> }
begin
  E:= AYaccStackElement(StackElement);
  E.Value.Deriv:=nil;
  E.Value.Body:='';
  Result:=E
end
```

```
;

AYaccDerivation:
{ Derivs -> Derived CompNameList }
begin
 E:= AYaccStackElement(StackElement);
 E1:= AYaccStackElement(ElementFromProduction(1));
 E.Value.Deriv:=TStringList.Create;
 E.Value.Deriv:=E1.Value.Deriv;
 E.Value.Body:=' Derived from '+E1.Value.Body;
 Result:=E
end
;

AYaccOneCompName:
{ CompNameList -> SocAgentName }
begin
 E:= AYaccStackElement(StackElement);
 E0:= AYaccStackElement(ElementFromProduction(0));
 E.Value.Deriv:=TStringList.Create;
 E.Value.Deriv.Add(E0.Value.Name);
 E.Value.Body:=E0.Value.Name;
 Result:=E
end
;

AYaccManyCompNames:
{ CompNameList -> SocAgentName Comma CompNameList }
begin
 E:= AYaccStackElement(StackElement);
 E0:= AYaccStackElement(ElementFromProduction(0));
 E2:= AYaccStackElement(ElementFromProduction(2));
 E.Value:=E2.Value;
 E.Value.Body:=E0.Value.Name+', '+E.Value.Body;
 E.Value.Deriv.Insert(0,E0.Value.Name);
 Result:=E
end
;

AYaccNoExportation:
{ Exports -> }
begin
 E:= AYaccStackElement(StackElement);
 E.Value.Exprt:=nil;
 E.Value.Body:='';
 Result:=E
end
;

AYaccExportation:
{ Exports -> RightArrow CompNameList }
begin
 E:= AYaccStackElement(StackElement);
 E1:= AYaccStackElement(ElementFromProduction(1));
 E.Value.Exprt:=TStringList.Create;
 E.Value.Exprt:=E1.Value.Deriv;
 E.Value.Body:=' --> '+E1.Value.Body;
 Result:=E
end
;

AYaccSetSeqInstComp:
{ Comp -> Identifier CompType Identifier }
```

```
begin
  E:= AYaccStackElement(StackElement);
  E0:= AYaccStackElement(ElementFromProduction(0));
  E1:= AYaccStackElement(ElementFromProduction(1));
  E2:= AYaccStackElement(ElementFromProduction(2));
  E.Value.Name:=StrPas(E0.Lexeme.Buffer);
  E.Value.OfType1:=StrPas(E2.Lexeme.Buffer);
  E.Value.CompType:=E1.Value.CompType;
  E.Value.Body:=E.Value.Name+' : '+E1.Value.Body+' '+E.Value.OfType1;
  Result:=E;
end
;

AYaccTableComp:
{ Comp -> Identifier TableComp Identifier Index Identifier }
begin
  E:= AYaccStackElement(StackElement);
  E0:= AYaccStackElement(ElementFromProduction(0));
  E2:= AYaccStackElement(ElementFromProduction(2));
  E4:= AYaccStackElement(ElementFromProduction(4));
  E.Value.Name:=StrPas(E0.Lexeme.Buffer);
  E.Value.OfType1:=StrPas(E4.Lexeme.Buffer);
  E.Value.OfType2:=StrPas(E2.Lexeme.Buffer);
  E.Value.CompType:=4;
  E.Value.Body:=E.Value.Name+' : Table of '+E.Value.OfType2+' Indexed by '+E.Value.OfType1;
  Result:=E;
end
;

AYaccSetComp:
{ CompType -> SetComp }
begin
  E:= AYaccStackElement(StackElement);
  E.Value.CompType:=1;
  E.Value.Body:='Set of ';
  Result:=E;
end
;

AYaccSeqComp:
{ CompType -> SeqComp }
begin
  E:= AYaccStackElement(StackElement);
  E.Value.CompType:=2;
  E.Value.Body:='Sequence of ';
  Result:=E;
end
;

AYaccInstanceComp:
{ CompType -> InstanceComp }
begin
  E:= AYaccStackElement(StackElement);
  E.Value.CompType:=3;
  E.Value.Body:='Instance of ';
  Result:=E;
end
;

AYaccActionNul:
{ ActionsDec -> Actions }
begin
  E:= AYaccStackElement(StackElement);
```

```
  E.Value.Actions:=nil;
  Result:=E
end
;

AYaccActionDecl:
{ ActionDecl -> Actions ActionList }
begin
  E1:= AYaccStackElement(ElementFromProduction(1));
  Result:=E1
end
;

AYaccOneAction:
{ ActionList -> ActionDec }
begin
  E:= AYaccStackElement(StackElement);
  E0:= AYaccStackElement(ElementFromProduction(0));
  E.Value.Actions:=TList.Create;
  New(PE);
  PE^:=E0.Value;
  PE^.Typ:=14;
  E.Value.Actions.Add(PE);
  Result:=E
end
;

AYaccActionList:
{ ActionList -> ActionDec ActionList }
begin
  E:= AYaccStackElement(StackElement);
  E0:= AYaccStackElement(ElementFromProduction(0));
  E1:= AYaccStackElement(ElementFromProduction(1));
  E.Value:=E1.Value;
  New(PE);
  PE^:=E0.Value;
  PE^.Typ:=14;
  E.Value.Actions.Insert(0,PE);
  Result:=E
end
;

AYaccStarActionDecNo:
{ ActionDec -> Star Identifier ActionBody Exports }
begin
  E:= AYaccStackElement(StackElement);
  E1:= AYaccStackElement(ElementFromProduction(1));
  E2:= AYaccStackElement(ElementFromProduction(2));
  E3:= AYaccStackElement(ElementFromProduction(3));
  New(PE);
  with E.Value do
  begin
    Name:='*'+StrPas(E1.Lexeme.Buffer);
    Args:=E2.Value.Args;
    AExprt:=E3.Value.Exprt;
    Body:=Name+' '+E2.Value.Body+E3.Value.Body;
  end;
  Result:=E
end
;

AYaccActionDecNo:
{ ActionDec -> Identifier ActionBody Exports }
```

```
begin
  E:= AYaccStackElement(StackElement);
  E1:= AYaccStackElement(ElementFromProduction(0));
  E2:= AYaccStackElement(ElementFromProduction(1));
  E3:= AYaccStackElement(ElementFromProduction(2));
  New(PE);
  with E.Value do
  begin
    Args:=TStringList.Create;
    Name:=StrPas(E1.Lexeme.Buffer);
    Args:=E1.Value.Args;
    AExprt:=E2.Value.Exprt;
    Body:=Name+' '+E1.Value.Body+E2.Value.Body;
  end;
  Result:=E
end
;

AYaccStarActionDec:
{ ActionDec -> Comment Star Identifier ActionBody Exports }
begin
  E:= AYaccStackElement(StackElement);
  E0:= AYaccStackElement(ElementFromProduction(0));
  E4:= AYaccStackElement(ElementFromProduction(4));
  E2:= AYaccStackElement(ElementFromProduction(2));
  E3:= AYaccStackElement(ElementFromProduction(3));
  New(PE);
  with E.Value do
  begin
    Name:='*'+StrPas(E2.Lexeme.Buffer);
    Args:=E3.Value.Args;
    AExprt:=E4.Value.Exprt;
    Body:=Name+' '+E3.Value.Body+E4.Value.Body;
    Man:=E0.Value.Man
  end;
  Result:=E
end
;

AYaccActionDec:
{ ActionDec -> Comment Identifier ActionBody Exports }
begin
  E:= AYaccStackElement(StackElement);
  E0:= AYaccStackElement(ElementFromProduction(0));
  E1:= AYaccStackElement(ElementFromProduction(1));
  E2:= AYaccStackElement(ElementFromProduction(2));
  E3:= AYaccStackElement(ElementFromProduction(3));
  New(PE);
  with E.Value do
  begin
    Name:='*'+StrPas(E1.Lexeme.Buffer);
    Args:=E2.Value.Args;
    AExprt:=E3.Value.Exprt;
    Body:=Name+' '+E2.Value.Body+E3.Value.Body;
    Man:=E0.Value.Man
  end;
  Result:=E
end
;

AYaccActionNoParams:
{ ActionBody -> }
begin
```

```
      E:= AYaccStackElement(StackElement);
      with E.Value do
      begin
        Args:=nil;
        Body:='';
      end;
      Result:=E
    end
    ;

    AYaccActionParams:
    { ActionBody -> ( TypeList ) }
    begin
      E:= AYaccStackElement(StackElement);
      E1:= AYaccStackElement(ElementFromProduction(1));
      with E.Value do
      begin
        Args:=TStringList.Create;
        for i:=0 to E1.Value.Value.Count-1 do
        begin
          PE:=E1.Value.Value[i];
          Args.Add(PE^.Name);
        end;
        Body:='('+E1.Value.Body+')';
      end;
      Result:=E
    end
    ;

  else
    { Bad Production }
    ;
  end;
  if Result=nil then
    Result := StackElement;
end;
end.
```